VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV MIKROELEKTRONIKY
DEPARTMENT OF MICROELECTRONICS

IMPLEMENTACE PROCESORU MICROBLAZE V JAZYCE CODAL

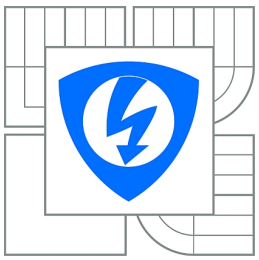
MICROBLAZE PROCESSOR IMPLEMENTATION USING CODAL LANGUAGE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR
Bc. Radek Hájek

VEDOUCÍ PRÁCE
SUPERVISOR
Ing. Marián Pristach

BRNO 2016



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav mikroelektroniky

Diplomová práce

magisterský navazující studijní obor
Mikroelektronika

Student:	Bc. Radek Hájek	***ID:***	146824
Ročník:	2	***Akademický rok:***	2015/2016

NÁZEV TÉMATU:

Implementace procesoru MicroBlaze v jazyce CodAL

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s architekturou procesoru Xilinx MicroBlaze. Podle dostupných informací navrhněte vhodnou mikroarchitekturu procesoru s instrukční sadou kompatibilní s procesorem MicroBlaze. Navržený procesor popište v jazyce CodAL na instrukční úrovni a následně na RTL úrovni. Správnou funkci procesoru ověřte na programech ze standardní testovací sady.
Navrhněte platformu s procesorem a vytvořte aplikaci, která bude demonstrovat funkci procesoru.

DOPORUČENÁ LITERATURA:

Podle pokynů vedoucího práce

Termín zadání:	8.2.2016	***Termín odevzdání:***	26.5.2016

Vedoucí práce: Ing. Marián Pristach
Konzultanti diplomové práce:

doc. Ing. Lukáš Fujcik, Ph.D.
Předseda oborové rady

ABSTRAKT

Diplomová práce obsahuje teoretický základ, rozdělení a funkce procesorů. Shrnuje princip zřetězeného zpracování instrukcí a druhy hazardů v mikroarchitektuře procesorů. Dále seznamuje s možnostmi návrhu procesorů pomocí jazyku CodAL, který je vyvíjen firmou Codasip. V praktické části práce byl vytvořen model procesoru MicroBlaze od firmy Xilinx v jazyce CodAL. Navržený model byl otestován a implementován do obvodu FPGA v rámci praktické ukázky.

KLÍČOVÁ SLOVA

Procesor, CodAL, MicroBlaze, zřetězené zpracování, hazardy

ABSTRACT

The diploma thesis contains theoretical basis, classification and function of processors. It summarizes the principle of pipelined instruction processing and the types of hazards in the microarchitecture of the processor. It also introduces design of processors using CodAL language developed by Codasip company. In the practical part of the thesis the model of MicroBlaze core developed by Xilinx company was described in the CodAL language. Designed model was tested and implemented into the FPGA device as practical example.

KEYWORDS

Processor, CodAL, MicroBlaze, pipelining, hazards

HÁJEK, R. *Implementace procesoru MicroBlaze v jazyce CodAL*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2016. 64 s. Vedoucí diplomové práce Ing. Marián Pristach, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Implementace procesoru MicroBlaze v jazyce CodAL“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing. Mariánu Pristachovi, Ph.D. za odborné vedení, trpělivost a podnětné návrhy k práci. Dále bych rád poděkoval slečně Lence Hermanové za podporu nejen při zpracování této práce. Poděkování patří také firmě Codasip za možnost zpracování diplomové práce.

Brno ....

podpis autora

SIX research centre
sensor, information and communication systems

Faculty of Electrical Engineering
and Communication

Brno University of Technology
Technicka 12, CZ-61600 Brno, Czechia

http://www.six.feec.vutbr.cz

Experimentální část této diplomové práce byla realizována na výzkumné infrastruktuře vybudované v rámci projektu CZ.1.05/2.1.00/03.0072
Centrum senzorických, informačních a komunikačních systémů (SIX)
operačního programu Výzkum a vývoj pro inovace.

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

2007–13
OP Výzkum a vývoj pro inovace

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

ÚVOD

V dnešní době dochází k neustálému vývoji výroby integrovaných obvodů a přechází se na stále menší velikosti struktur. Přitom hnací silou tohoto vývoje jsou právě číslicové systémy, zejména však procesory. Každoročně je vyrobeno přes miliardu procesorů. Jejich využití je univerzální a proto jsou přítomny téměř ve všech systémech. Ukazuje se, že v některých jednoúčelových systémech nejsou kladeny požadavky na univerzálnost, více se však dbá na velikost a spotřebu čipu, případně výpočetní výkon na konkrétní aplikaci. Příkladem může být rozvíjející se nositelná elektronika nebo tzv. internet věcí.

Tyto specifické požadavky lze částečně splnit například použitím aplikačně specifických procesorů (ASIP – Application-Specific Instruction Set Processor). V případě výrobců FPGA jsou často k dispozici procesory ve formě IP (Intelectual Property) jader. Tyto procesory jsou dodávány v několika verzích, kde každá verze je optimalizovaná pro specifický požadavek (rychlost, plocha, spotřeba). Výrobci navíc často umožňují pouhým nastavením přepínače modifikaci vnitřní struktury procesoru - použití speciálních instrukcí nebo hardwarových bloků. Příkladem takového procesoru je i MicroBlaze od firmy Xilinx. Tyto procesory nabízejí pouze omezenou optimalizaci pro konkrétní aplikaci. Vyšší optimalizaci lze dosáhnout návrhem procesoru s aplikačně specifickou instrukční sadou. Návrh takového procesoru může být oproti použití hotového procesoru zdlouhavý, avšak může vést k výrazně lepším výsledkům a to ve všech požadavcích.

Na návrh procesorů s aplikačně specifickou instrukční sadou se specializuje i brněnská společnost Codasip. Díky nástrojům firmy Codasip (Codasip Studio) lze dobu vývoje zcela nového aplikačně specifického procesoru zkrátit na několik týdnů, což může výrazně urychlit uvedení nového výrobku na trh. Navíc lze vyzkoušet několik různých verzí v extrémně krátké době a lze tedy dosáhnout optimálního řešení.

Tato diplomová práce se zabývá návrhem jádra procesoru MicroBlaze od firmy Xilinx v nástrojích firmy Codasip. Navržené jádro procesoru není určeno pro komerční použití z důvodu licenčních podmínek. Návrh procesoru v rámci této práce bude sloužit výhradně ke srovnání výsledků dosažených použitím Codasip Studia s řešením poskytovaným firmou Xilinx.

1 TEORETICKÁ ČÁST

1.1 Procesory

Procesor (Central Processing Unit - CPU), někdy též označovaný jako mikroprocesor, je programovatelný logický obvod, který vykonává sled instrukcí programu uloženého v paměti. Změnou programu lze měnit funkci realizovanou procesorem, proto se procesory používají v téměř každém elektronickém zařízení. Každý procesor lze charakterizovat několika parametry, z nichž asi nejdůležitější je rychlost (výpočetní výkon) vyjadřovaná jako počet operací provedených za jednu sekundu. Protože rychlost dnešních procesorů je obrovská, používaná jednotka rychlosti jádra je milion instrukcí za sekundu MIPS (Million Instructions Per Second).

1.1.1 Architektura procesorů

Pod pojmem architektura procesorů rozumíme obecné uspořádání procesoru, přitom však nezáleží na konkrétní realizaci na úrovni čipu. Architektura procesoru je definována instrukční sadou (seznam všech instrukcí, které jsou procesoru známy), vnitřními zdroji procesoru (registry, paměti), šířkou sběrnice atd. S ohledem na instrukční sadu je architektura někdy nazývána jako ISA - Instruction Set Architecture. Procesory, které patří do stejné architektury, jsou spolu vzájemně kompatibilní - používají společný základ instrukcí, které jsou kódované stejným způsobem.

Architektury lze dělit podle mnoha kritérií. Jedním z kritérii může být délka slova (zjednodušeně šířka datové a paměťové sběrnice). Čím větší je délka slova, tím je procesor schopen pracovat s většími čísly a může rychleji zpracovávat data. Dalším dělením může být podle toho, zda je oddělena paměť programu a dat (Harvardská architektura), případně je tato paměť společná (von Neumannova)[1].

Dle instrukční sady můžeme procesory rozdělit na 2 typy:

1. s kompletní instrukční sadou (CISC - Complete Instruction Set Computing),
2. s redukovanou instrukční sadou (RISC - Reduced Instruction Set Computing).

Procesory s kompletní instrukční sadou mají mnoho instrukcí, které realizují mnoho různých a poměrně složitých funkcí, typické jsou instrukce pro různé adresní módy. Instrukce mají proměnnou délku i dobu vykonávání v procesoru. Výhodou kompletní instrukční sady je zvýšení výkonu procesoru tím, že se některé operace mohou provádět paralelně, případně se snižuje počet přístupů do paměti. Instrukce jsou však často velmi specifické, a proto nejsou používány příliš často, navíc se jejich složitost projevuje i na velikosti hardware.

Opačný přístup používají procesory s redukovanou instrukční sadou, které vychází z předpokladu, že většinu použitých instrukcí tvoří pouze malá sada některých instrukcí - ukládání a čtení z paměti, podmíněné skoky, porovnání aj. Procesory používají menší počet obecnějších instrukcí, složitější instrukce jsou skládány z obecnějších. Z toho plynou větší nároky na velikost programové paměti, avšak složitost hardware se snižuje, což vede opět ke zvýšení výkonu. RISC architektura s sebou nese několik výrazných vlastností:

- malý instrukční soubor konstantní délky s málo adresními módy,
- vyšší počet registrů, přístup do paměti výhradně přes instrukce load a store,
- v každém taktu hodin by měla být dokončena alespoň jedna instrukce,
- zjednodušení hardware, optimalizace překladačem.

Dnešní procesory v sobě většinou kombinují prvky RISC a CISC architektury a nelze je jednoznačně dělit, protože jde vždy o kompromis mezi oběma směry [2].

1.1.2 Mikroarchitektura procesorů

Architektura procesoru nabízí jen abstraktní pohled na procesor. Konkrétní pohled do vnitřního uspořádání procesoru nabízí mikroarchitektura, která určuje vnitřní uspořádání na nižší úrovni abstrakce. Je běžné, že k jedné architektuře existuje několik různých mikroarchitektur. Tím lze dosáhnout optimalizace daného procesoru na konkrétní požadavek (např. na plochu čipu, rychlost, spotřebu aj.), případně se liší procesory dodávané různými výrobci (např. architektura x86 od Intelu či AMD). Mezi těmito procesory je však zachována kompatibilita a přenositelnost kódu [3].

1.2 Výkon procesoru

Protože se zvyšují požadavky na rychlost (výpočetní výkon) procesorů, moderní procesory pracují s určitým druhem paralelismu. Procesory můžeme rozdělit podle způsobu zpracování instrukcí a dat (úrovně paralelismu) na skalární, superskalární, s dlouhým instrukčním slovem (VLIW - Very Long Instruction Word), vektorové (SIMD - Single Instruction Multiple Data) aj. Paralelismus může být na úrovni instrukcí (např. skalární, superskalární, VLIW) či úrovni datové (vektorové procesory). V pokročilých systémech lze používat i vícevláknové nebo vícejádrové procesory. Tato práci se bude zabývat skalárními procesory [4] [5].

1.2.1 Zřetězené zpracování

Důležitým požadavkem na procesory (i jiné digitální systémy) je zejména pracovní frekvence, která je dána nejdelší kombinační cestou mezi dvěma registry. Pracovní kmitočet lze zvyšovat rozdělením kombinační cesty jedním či více registry, samotný výpočet pak ale trvá více hodinových taktů.

Ve skalárních procesorech se používá zřetězené zpracování instrukcí (pipelining). Vykonávání instrukcí se rozdělí na několik stupňů (stages) oddělených registry, každá instrukce pak prochází těmito stupni postupně tak, že v každém stupni je právě jedna instrukce. Obdobný princip je používán ve výrobních linkách. Tímto principem je přirozeně rozdělena i kombinační cesta a zvyšuje se pracovní frekvence procesoru a společně s ní i počet zpracovaných instrukcí v čase.

Na obrázku 1.1 je v časovém diagramu zobrazeno zpracování instrukcí v nezřetězené lince (a), kde je výpočet instrukce rozdělen na 3 části a trvá 3 hodinové takty, a v 3 stupňové zřetězené lince (b). Zatímco bez použití zřetězeného zpracování (a) jsou 2 instrukce zpracovány za šest hodinových taktů, při použití zřetězeného zpracování (b) jsou během 6 hodinových cyklů dokončeny čtyři instrukce a další dvě jsou v lince již rozpracovány [6].

a) nezřetězené zpracování

hodinový takt	část 1	část 2	část 3
1	i1		
2		i1	
3			i1
4	i2		
5		i2	
6			i2

b) zřetězené zpracování

hodinový takt	stupeň 1	stupeň 2	stupeň 3
1	i1		
2	i2	i1	
3	i3	i2	i1
4	i4	i3	i2
5	i5	i4	i3
6	i6	i5	i4

Obr. 1.1: Zpracování instrukcí v nezřetězené a zřetězené lince

Při zřetězeném zpracování lze mít rozpracováno několik instrukcí současně. Počet těchto instrukcí je dán počtem stupňů dané linky (3 instrukce na příkladu z obrázku 1.1), v ideálním případě je každý takt dokončena jedna instrukce. Pro výkon je důležité, aby výpočet v každém z těchto stupňů trval přibližně stejnou dobu, protože mezní kmitočet je dán právě nejdelším z těchto časů. Při optimálním rozvržení linky přináší zřetězené zpracování výrazné navýšení výkonu [7].

1.2.2 Hazardy

V reálném procesoru nelze docílit ideálního zřetězeného zpracování kvůli omezení v mikroarchitektuře [6]. Vlivem těchto omezení (hazardů) a jejich řešením je snižován teoretický mezní výkon procesoru. Hazardy můžeme rozdělit na:

1. datové,
2. řídící,
3. strukturální.

1.2.3 Datové hazardy

Datové hazardy vznikají vlivem závislostí mezi instrukcemi. Můžeme rozlišit několik typů datových hazardů:

1. RAW (Read After Write) hazard,
2. WAR (Write After Read) hazard,
3. WAW (Write After Write) hazard.

RAW hazard je nejběžnější z hazardů. Vzniká, pokud instrukce potřebuje výsledek předcházející instrukce, který však ještě nebyl zapsán, a instrukce tedy nemá aktuální data. To je způsobeno tím, že zápis do registrů probíhá na konci linky, zatímco čtení na začátku linky. Čím má linka více stupňů, tím hazard trvá déle. Na příkladu je vidět, že instrukce i2 používá registr R2, který je používán jako cílový registr předcházející instrukce.

```
i1: ADD R2, R1, R3  // R2 = R1 + R3
i2: ADD R4, R2, R3  // R4 = R2 + R3
```

WAR hazard vzniká tehdy, je-li výsledek instrukce zapsán dříve než některá z předcházejících instrukcí stihla přečíst data. Následující instrukce tedy přepsala data, která jsou ještě třeba. Tento typ hazardu vzniká pouze u procesorů, které mají zpracování instrukcí mimo pořadí (OOO - Out Of Order). Instrukce tedy nejsou zpracovány přesně v pořadí, v jakém jsou uvedeny v programu. Hazard WAR by vznikl, kdyby instrukce i2 byla dokončena ještě před vykonáním instrukce i1. Instrukce i1 by poté používala hodnotu registru R5, která neodpovídá toku programu.

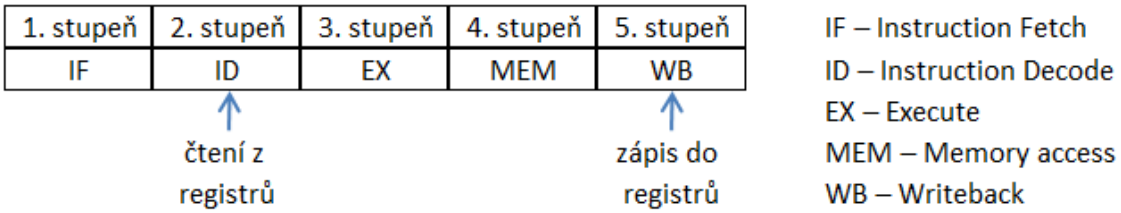
```
i1: ADD R4, R1, R5  // R4 = R1 + R5
i2: ADD R5, R1, R2  // R5 = R1 + R2
```

WAW hazard se opět projevuje pouze při OOO zpracování tak, že instrukce i2 by byla dokončena dříve než instrukce i1. V registru R2 by tedy byla uložena data odpovídající instrukci i1, ačkoli ve skutečnosti měla být přepsána instrukcí i2.

```
i1: ADD R2, R4, R7  // R2 = R4 + R7
i2: ADD R2, R1, R3  // R2 = R1 + R3
```

Tato práce se bude podrobně zabývat pouze hazardy typu RAW, protože ostatní typy hazardů se projevují pouze při zpracování mimo pořadí, které není obsahem této práce. Na obrázku 1.2 je příklad 5-stupňové mikroarchitektury. Jednotlivé stupně jsou:

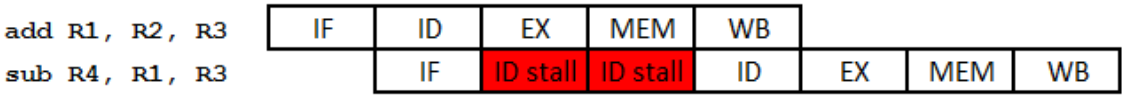
1. IF (Instruction Fetch) – načítání instrukce,
2. ID (Instruction Decode) – dekodování instrukce,
3. EX (Execute) – vykonávání instrukce,
4. MEM (Memory access) – přístup do paměti,
5. WB (Writeback) – zápis do registrového pole.



Obr. 1.2: Příklad 5-stupňové mikroarchitektury

Čtení registrů nastává v 2. stupni (ID), zatímco zápis až v 5. stupni (WB). RAW hazard nastane v případě, že je v lince zpracovávána dvojice po sobě jdoucích vzájemně závislých instrukcí, např.:

```
i1: ADD R1, R2, R3   // R1 = R2 + R3
i2: SUB R4, R1, R3   // R4 = R1 - R3
```



Obr. 1.3: Znázornění RAW hazardu v mikroarchitektuře

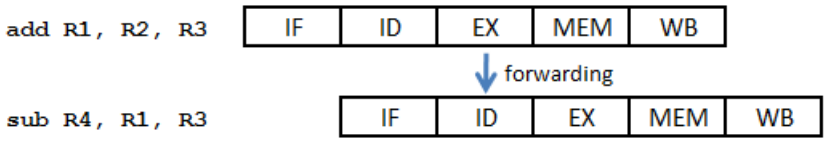
V tomto případě je třeba, aby instrukce `SUB` (i2) počkala ve stupni ID, dokud instrukce `ADD` (i1) neprojde až do stupně WB a její výsledek nebude zapsán do registru. Pro vyřešení datového hazardu je třeba pozastavit zpracování instrukce sub ve stupni ID na 2 takty - na obrázku 1.3 je tato situace naznačena jako „ID stall“. Na obrázku 1.4 je zobrazeno zpracování instrukcí v jednotlivých stupních linky v čase - lze vidět, že zatímco instrukce `SUB` čeká v ID stupni a instrukce `ADD` postupuje dále, do stupně EX je vkládána instrukce `NOP` (No-Operation).

V některých případech je však výsledek připraven mnohem dříve než je skutečně zapsán do registru. V takovém případě lze přeposlat data mezi stupněmi a ušetřit

hodinový takt	IF	ID	EX	MEM	WB
1	add				
2	sub	add			
3		sub	add		
4		sub	NOP	add	
5		sub	NOP	NOP	add
6			sub	NOP	NOP
7				sub	NOP
8					sub

Obr. 1.4: Znázornění RAW hazardu v čase

instrukci `sub` čekání (obrázek 1.5). Použití této „zkratky“ se říká přeposílání (forwarding nebo též register bypassing). Pokud zpracování trvá více taktů (např. u instrukce načítání z paměti `LD`), lze čekání eliminovat jen částečně a v některých případech může být třeba procesor pozastavit. Oba výše zmíněné přístupy vyžadují podporu v hardwaru.



Obr. 1.5: Odstranění RAW hazardu pomocí přeposílání (forwarding)

RAW hazardy lze však řešit i bez podpory hardwaru použitím inteligentních překladačů. Překladač má pro každou instrukci definované zpoždění (latenci), při překladu programu se snaží instrukce poskládat tak, aby na sobě v rámci definovaných latencí nezávisely. V případě, že v procesoru není implementováno přeposílání, překladač RAW hazard vyřeší přeskládáním instrukcí. Mezi instrukce i1 a i2 tedy vloží další instrukce. Pokud překladač nenajde žádné vhodné instrukce, zbaví se závislostí vložením instrukcí `NOP`. Tento přístup sice nevyžaduje podporu v mikroarchitektuře, avšak vkládání instrukcí `NOP` zvětšuje požadavky na velikost programu v paměti. Implementací přeposílání se latence snižuje.

Pro optimální výsledky lze oba principy zkombinovat tak, že překladač se snaží zbavit hazardů přeskládáním instrukcí vždy. Pokud není v mikroarchitektuře implementováno přeposílání nebo v některých případech nelze výsledek přeposlat (výpočet trvá více taktů), překladač odstraní hazardy vložením instrukcí `NOP`. Pokud je v procesoru implementováno přeposílání i detekce hazardů, vložení instrukcí `NOP`

není třeba. Z výše zmíněných příkladů je jasné, že optimální překladač potřebuje alespoň částečné údaje o použité mikroarchitektuře.

1.2.4 Řídící hazardy

Řídící hazardy jsou způsobeny instrukcemi skoků, zejména podmíněnými skoky. V ideálním případě je ihned po načtení instrukce známa adresa následující instrukce. Toho lze ve skutečné mikroarchitektuře dosáhnout jen velmi těžko, protože výpočet adresy a podmínky skoku trvá určitou dobu. Instrukce skoků proto mají vždy nějakou penalizaci. Jednou z možností, jak vyřešit řídící hazard, je pozastavení procesoru na dobu, než bude určena následující instrukce. Linka se pozastavuje u všech skoků, vykonaných i nevykonaných, což vede k velkým prodlevám v lince. Mnohem efektivnější je při načtení instrukce skoku pokračovat ve vykonávání programu tak, jako by se neskákalo. V případě, že se skutečně neskáče, není neprovedený skok nijak penalizován. V opačném případě je třeba načtené instrukce odstranit (clear) z linky (prakticky se jedná o nahrazení `NOP` instrukcí) a pokračuje se na adrese skoku s penalizací jako v předchozím případě. Průběh instrukcí při vykonaném i nevykonaném skoku, pokud je programový čítač (PC) změněn ve stupni MEM, je zobrazen na obrázku 1.6. Instrukce *in* na obrázku 1.6 jsou instrukce, které následují v programu skokovou instrukci, *ic* je instrukce cíle skoku. Vyčištění linky je naznačeno červenou barvou.

hodinový takt	FE	ID	EX	MEM	WB
1	skok				
2	in1	skok			
3	in2	in1	skok		
4	in3	in2	in1	skok	
5	in4	in3	in2	in1	skok
6	in5	in4	in3	in2	in1

a) nevykonaný skok

hodinový takt	FE	ID	EX	MEM	WB
1	skok				
2	in1	skok			
3	in2	in1	skok		
4	in3	in2	in1	skok	
5	NOP	NOP	NOP	NOP	skok
6	ic1	NOP	NOP	NOP	NOP

b) vykonaný skok

Obr. 1.6: Časový průběh instrukcí v lince u nevykonaného a vykonaného skoku

Protože instrukce skoku je v programech poměrně častá, má i malá penalizace velký vliv na efektivní vykonávání programu. Proto je snahou co nejvíce snižovat penalizaci, například úpravou mikroarchitektury tak, aby vyhodnocení skoku probíhalo ideálně ihned po načtení instrukce. Pro opravdu náročné aplikace lze použít

predikci skoků - jednotku, která na základě historie vykonávání skoků dokáže předpovídat vykonání skoku.

Dalším řešením, které nevyžaduje velké zásahy do mikroarchitektury, je použití tzv. skoků s „delay“ slotem. Penalizace se snižuje tak, že jedna nebo více instrukcí se po skoku vykoná vždy, ať už se skáče nebo ne. Po načtení instrukce skoku se pokračuje v načítání následujících instrukcí stejně jako v dříve diskutovaném řešení. V případě, že se skok vykoná, nedochází ke zrušení všech instrukcí mezi instrukcí skoku a nově načtenou instrukcí jako na obrázku 1.6. V případě skoků s „delay“ sloty jsou některé instrukce v lince zachovány. Snížení penalizace je v tomto případě závislé na tom, jak efektivně dokáže překladač plnit „delay“ sloty [8].

1.2.5 Strukturální hazardy

Posledním typem hazardů jsou strukturální hazardy. Tyto hazardy vznikají v důsledku kolizí na sdílených zdrojích v mikroarchitektuře - typicky pamětí, sekvenční děličky, aj. Základem pro eliminaci strukturálních hazardů je návrh vhodné mikroarchitektury tak, že se ke zdrojům přistupuje z jednoho místa (příkladem může být zápis do registrového pole z posledního stupně linky). Strukturální hazardy lze opět odstranit přeskládáním instrukcí v překladači či detekcí a pozastavením v hardwaru. Jediné další řešení je replikace zdrojů (více děliček, přidání portů u pamětí atd.). Toto řešení není vždy vhodné hlavně kvůli nárokům na plochu vyrobeného čipu [9].

1.3 Možnosti návrhu procesorů

Neustále se zlepšující technologie a klesající cena výroby v polovodičovém průmyslu umožňuje obrovský rozvoj integrovaných obvodů. Stále více se prosazuje výroba systémů na čipu (SoC), které často integrují až několik procesorů, paměti a periferie na jediném čipu, optimalizovaných a navržených pro specifickou funkci. Neustále tedy rostou požadavky na návrháře - systémy nabývají na složitosti, ale nové produkty musí být uvedeny na trh v co nejkratším čase. Požadovaný čas na návrh, popis i verifikaci systémů se snižuje. Proto se pro jejich návrh vyvíjí nové nástroje, které umožňují dosáhnout vysoké úrovně automatizace.

1.3.1 Jazyky pro návrh procesorů

Jednou z možností je použití jazyků pro popis hardware (HDL jazyků). Nejčastěji používané HDL jazyky jsou VHDL a Verilog. Tyto jazyky jsou sice vhodné pro návrh na úrovni hradel, případně simulaci, pro návrh procesorů však nejsou

vhodné, protože neumožňují popis dalších důležitých nástrojů pro programování navrženého procesoru (SW nástroje) - assembleru, překladače, ladícího nástroje (debugger) a dalších.

Simulace dlouhých programů může být na úrovni HDL jazyků zdlouhavá. Proto se pro návrh procesorů používají jazyky pro popis architektur (ADL). Tyto jazyky lze rozdělit do několika kategorií: [10]

1. jazyky zaměřené na instrukční sadu: ML, ISDL, Valen-C a CSDL,
2. jazyky zaměřené na strukturu: MIMOLA, AIDL,
3. smíšené jazyky: FlexWare, Mdes, PEAS, RADL, LISA a CodAL.

Jazyky zaměřené na instrukční sadu jsou většinou určené ke generovaní překladačů vyšších programovacích jazyků. Obsahují informace o instrukční sadě, sekvencích a latencích instrukcí, ale neobsahují žádné informace o mikroarchitektuře.

Strukturální jazyky používají popis mikroarchitektury i softwarové části. Mikroarchitektura a její propojení je popsána na nízké úrovni abstrakce. Tyto jazyky jsou použitelné jak pro syntézu (generování HDL), tak generování některých SW nástrojů. Vzhledem k nízké úrovni abstrakce popisu architektury je nevýhodou nízká rychlost generovaných simulátorů.

Poslední skupinou jsou smíšené jazyky pro popis instrukční sady i struktury zároveň, které jsou kombinací obou předchozích. Výhodou je, že umožňují generování všech nástrojů a to v optimální míře. Příkladem můžou být generované simulátory, které jsou několikanásobně rychlejší než simulace HDL jazyků. Mezi tyto jazyky patří i jazyk CodAL, jenž byl použit pro návrh procesoru v rámci této práce.

1.3.2 Jazyk CodAL

Jazyk CodAL (Codasip Architecture Language) byl vyvinut na Fakultě informačních technologií na VUT v Brně v rámci projektu Lissom (Language for Instruction Set Simulator-Oriented Model) a v současnosti je dále rozvíjen společností Codasip. Slouží zejména k návrhu a prototypování procesorů se specifickou instrukční sadou (ASIP) a to v návrhovém prostředí Codasip Studio. Z modelu popsaného v jazyce CodAL je možné generovat nástroje pro programování a simulace, ale také RTL popis procesoru v jazyce VHDL nebo Verilog. Je vhodný k souběžnému návrhu hardwaru a softwaru, díky vysoké míře automatizace umožňuje rychlé změny v instrukční sadě i mikroarchitektuře procesoru, rychlé testování různých možností a optimalizací.

Model procesoru může být popsán na dvou úrovních:

1. model na instrukční úrovni – IA model (Instruction Accurate model),
2. model na RTL úrovni – CA model (Cycle Accurate model).

Model na instrukční úrovni popisuje chování procesoru bez časových závislostí. To znamená, že všechny události v modelu jsou obsluhovány okamžitě bez časového zpoždění. Data jsou v paměti vždy připraveny ihned po požádání, zápis do registrů probíhá okamžitě atd. Slouží zejména k popisu instrukční sady, ze které se generují nástroje jako je assembler a překladač jazyka C. Je také možnost vygenerovat simulátor na úrovni instrukcí (IA simulátor), který je vhodný pro testování překladače a ladění programů.

Model na RTL úrovni popisuje skutečnou mikroarchitekturu procesoru. Jedná se o model, který přesně respektuje časování a slouží ke generování RTL popisu procesoru v jednom z HDL jazyků. Pro odladění chyb v návrhu je také možné vygenerovat CA simulátor, který odpovídá HDL popisu, ale jeho simulace je několikanásobně rychlejší než simulace na úrovni HDL.

Pro verifikaci návrhu se používá kombinace obou modelů. IA model slouží jako referenční model k modelu na RTL úrovni. Každý model je rozdělen na 2 části - popis samotného jádra procesoru (ASIPu) a popis platformy. Platforma obsahuje paměti, periferie procesoru a instance jádra procesoru a je pro obě úrovně modelu společná. Toto rozdělení umožňuje definici platformy s více jádry procesoru [11] [12].

Popis modelů se skládá z několika částí:

1. definice zdrojů a rozhraní,
2. popis instrukční sady a dekodéru instrukcí,
3. popis funkčních jednotek.

1.3.3 Definice zdrojů a rozhraní

V rámci definic zdrojů jsou definovány všechny registry a signály v procesoru. Signály představují propojení v procesoru na úrovni vodičů. V simulaci jsou všechny signály na začátku každého hodinového taktu vynulovány, zápis do signálů probíhá okamžitě a je tedy možné ihned číst novou hodnotu. V modelu na RTL úrovni registry představují D klopné obvody, zapsaná hodnota se v registru objeví až následující hodinový takt. V modelu na instrukční úrovni probíhá zápis do registrů okamžitě. Registry lze dělit na registry architekturální a nearchitekturální. Architekturální registry jsou registry, které jsou přístupné z pohledu programátora a jsou označované klíčovým slovem `arch`. Příkladem může být registrové pole, stavový registr aj. Speciálním architekturálním registrem je programový čítač (PC - Program Counter) označovaný klíčovým slovem `pc`. Tento registr musí být v modelovaném procesoru vždy.

Registry, které nemají klíčové slovo `arch`, jsou považované za nearchitekturální registry uvnitř procesoru, které slouží například jako oddělovací registry mezi jednotlivými stupni zřetězené linky. Tyto registry slouží zejména pro model na RTL úrovni

a programátor s nimi přímo nepracuje. V následujícím příkladu je uvedena definice některých zdrojů procesoru. Konkrétně se jedná o definici registru PC (`r_pc`), architekturálního registru `r_MSR`, nearchitekturální registru (`r_ex_imm`), signálu s výsledkem aritmeticko-logické operace (`s_ex_result_alu`) a registrového pole (`rf_gp`). Klíčová slova pro definice jsou `register`, `register_file` a `signal`; `bit[]` určuje bitovou šířku daného zdroje. U registrového pole je třeba definovat počet registrů (`size`) a počet portů pro čtení a zápis (`dataports`). Registry mohou mít navíc uvedenu reset hodnotu a do kterého stupně linky patří [13].

```
// programovy citac
pc register   bit[32] r_pc;
// stavovy registr
arch register bit[32] r_MSR;
// vysledek ALU operace - signal
signal        bit[32] s_ex_result_alu;
// nearchitekturalni registr pro 16-bitovou konstantu
register      bit[16] r_ex_imm;

// 32x32-bitove registrove pole
register_file bit[32] rf_gpr
{
    size = 32;
    dataports = { 3, 1 }; // porty - cteni/zapis
};
```

Další důležitou součástí modelu je definice rozhraní procesoru, tedy sběrnic a portů pro komunikaci s okolím, například s pamětmi. V následujícím příkladu je uvedena definice sběrnice pro programovou paměť a portů procesoru. Sběrnice je definována několika parametry:

1. `bits` - určuje šířku adresy, slova v paměti a nejmenší adresovatelné jednotky (LAU - Least Addressable Unit)
2. `type` - typ sběrnice: MEMORY / CLB (Codasip local bus); MASTER / SLAVE
3. `flag` - čtení (R), zápis (W) nebo čtení i zápis (RW)
4. `endianness` - endianita BIG/ LITTLE

```
interface ibus
{
    bits = { 32, 32, 8 }; //sirka adresy/sirka slova/sirka LAU
    type = CLB:MASTER;
    flag = R;             // pouze pro cteni
    endianness = BIG;
};
```

```
port bit[1]    p_irq;      // port pro preruseni
```

1.3.4 Popis instrukční sady a dekodéru instrukcí

Po definici zdrojů a rozhraní procesoru je nutné definovat instrukční sadu pro IA model, respektive dekodér instrukcí pro CA model. Popis probíhá pomocí objektů, které lze různě spojovat a kombinovat. Tyto objekty jsou v jazyce CodAL nazývány klíčovým slovem `element`. Každý element je rozdělen na další sekce:

1. `assembler`,
2. `binary`,
3. `semantics`,
4. `return`.

`Assembler` sekce uvádí základní textový popis daného elementu a odráží popis instrukce nebo její části v jazyce assembler. Sekce `binary` obsahuje informaci o binárním kódování daného elementu. `Semantics` určuje chování instrukce či elementu při vykonání v simulátoru, v IA modelu je sémantika instrukcí také extrahována při generování překladače. V tomto procesu jsou referenční instrukce překladače spojeny s konkrétními instrukcemi daného procesoru. Poslední sekcí je `return`, která se používá jako referenční informace při spojování více elementů. Jednotlivé elementy lze poté spojovat do skupin pomocí klíčového slova `set`. V následujícím příkladě je ukázáno postupné modelovaní instrukční sady procesoru. Modelování začíná na úrovni jednoduchých elementů, které jsou postupně skládány dohromady. Nejprve je vytvořen element reprezentující jediný registr z registrového pole. Tento registr bude v jazyku assembler nazýván jako `R1`, bude zakódován na 5 bitech jako 0b00001 a při použití v dalších elementech bude předávána přímo adresa registru - číslo 1.

```
element reg1
{
    assembler { "R1" };
    binary    { 1:bit[5] };
    return    { 1; };
};
```

Podobným způsobem můžeme vytvořit všechny ostatní registry, ze kterých uděláme jediný element pomocí příkazu `set`.

```
// tento set slucuje vsechny registry z reg. pole
set gpreg += reg1, reg2, ...;
```

S takto vytvořenou skupinou můžeme přímo namodelovat instrukce, které používají pouze registrové operandy - například instrukci `ADD`. Konstrukce `use gpreg as` znamená, že se použije instance výše definovaného elementu s definovaným názvem. V tomto případě je použit hned třikrát. V uvedeném příkladu je namodelována instrukce „add reg, reg, reg“, kde `reg` je jakýkoli registr ze skupiny `gpreg`. V sekci `semantics` je definováno chování instrukce - jsou vyčteny hodnoty z registrového pole z adres, které vrací return sekce `reg_src1` a `reg_src2`, proveden součet a výsledek je zapsán zpět do registrového pole na adresu `reg_dst`.

```
element i_add
{
    use gpreg as reg_dst, reg_src1, reg_src2;

    ...

    semantics
    {
        int32 res, src1, src2;

        src1 = rf_gpr[reg_src1];
        src2 = rf_gpr[reg_src2];
        res = src1 + src2;
        rf_gpr[reg_dst] = res;
    };
};
```

Všechny popsané instrukce jsou pak spojeny do elementu `isa`, který představuje všechny instrukce procesoru:

```
set isa = i_add, ...;
```

Popis dekodéru instrukcí pro model na RTL úrovni probíhá podobným způsobem. V sekci `semantics` se však místo obecné funkce instrukce uvádí přímo nastavení signálů v mikroarchitektuře. Dekodér navíc také obvykle nepoužívá všechny bity instrukce, ale pouze část s operačními kódy.

1.3.5 Popis funkčních jednotek

Poslední částí modelu jsou funkční jednotky označované klíčovým slovem `event`. Protože CodAL je sekvenční jazyk a jednotlivé příkazy jsou vykonávány v pořadí, je nutné zajistit správné pořadí vykonávání příkazů. To je v modelu zaručeno právě pomocí funkčních jednotek, které jsou aktivovány v určitém pořadí. Tím lze jednoduše ovlivnit pořadí vykonávaných příkazů a docílit očekávané funkce.

Funkční jednotka může mít 3 sekce - `semantics`, `timing` a `decoders`. V sekci `semantics` je uvedeno, co se stane, pokud je daná funkční jednotka aktivována. V sekci `timing` jsou uvedeny další funkční jednotky, které jsou následně aktivovány. Sekce `decoders` je určena pro vložení dekodéru do simulace. Tím se specifikuje konkrétní signál, který je určený k dekódování.

Pro každý typ modelu jsou 2 povinné funkční jednotky – `main` a `reset`. Funkční jednotka `reset` specifikuje chování po resetu procesoru - obvykle se pouze nuluje registrové pole, protože registry mají nastavenou resetovací hodnotu přímo v jejich deklaraci atributem `default`. Funkční jednotka `reset` je vždy automaticky aktivována na začátku simulace. Funkční jednotka `main` je aktivována opakovaně na začátku každého taktu simulace, dokud nedojde k zastavení simulace.

IA model většinou používá pouze povinné funkční jednotky `reset` a `main`, protože vykonávání instrukce je popsáno přímo v sémantice instrukce a vykonává se okamžitě. V `main` je zajištěno čtení z instrukční paměti, aktualizace PC a volání dekodéru, přechod do obsluhy přerušení atd. Jednoduchá funkční jednotka `main` pro IA model je uvedena v následujícím příkladě.

```
event main // IA model
{
    use isa;

    semantics
    {
        // nacitani instrukce
        current_instr_reg = ibus[r_pc];

        // inkrementace PC o 4 po nacteni instrukce
        r_pc += 4;
    };
};
```

Pro CA model funkční jednotka `main` slouží zejména k aktivaci dalších funkčních jednotek v definovaném pořadí. V CA modelu je možné definovat linku procesoru a její stupně pomocí klíčového slova `pipeline`. Ve složených závorkách se pak postupně uvádí názvy jednotlivých stupňů linky.

```
// definice zretezene linky
pipeline pipe { IF, ID, EX, MEM, WB };
```

Jednotlivé funkční jednotky lze pak přiřazovat konkrétnímu stupni linky. Simulace pak probíhá podle pořadí aktivací funkčních jednotek v sekcích `timing`. Z `main` jsou aktivovány funkční jednotky, které odpovídají jednotlivým stupňům linky a z těchto jednotek mohou být aktivovány další funkční jednotky uvedené

v sekci `timing`. Aktivace funkčních jednotek může být i podmíněná, tedy některé funkční jednotky se mohou vykonávat jen za určitých podmínek. Někdy je kvůli přehlednosti vhodné funkční jednotky rozdělit na menší bloky. V následujícím příkladu je uvedena funkční jednotka `main` pro CA model, která pouze aktivuje další funkční jednotky odpovídající jednotlivým stupňům linky.

```
event main
{
    use fe;
    ...

    timing
    {
        fe();  // aktivuje funkcni jednotku fe
        ...    // aktivace dalsich funkcnich jednotek
    };
};
```

V další ukázce kódu je definována funkční jednotka `id`, která patří do ID stupně linky. Při její aktivaci je dokončeno načítání instrukce, vybrání části instrukce pro dekodér a aktivace další funkční jednotky `id_output`. Tímto způsobem lze zařídit, že ve funkční jednotce `id_output` budou platná data z aktuálně dekódované instrukce. Aktivací dojde k přepsání všech registrů. Předpokládá se však, že v následujícím stupni EX, který je z výše zmíněného principu aktivován dříve, došlo k přepsání všech potřebných údajů do jiných registrů.

```
event id
{

    use id_output;

    semantics
    {
        // dokonceni cteni dat na instrukcni sbernici
        ibus.ifinish(CP_FI_COMPLETE, s_id_instr);

        // 6 bitu pro dekoder [31..26]
        s_id_decode = (s_id_instr[31..26];
    };

    timing
    {
        id_output();
    };
};
```

1.3.6 Popis platformy

Jazyk CodAL je určený k popisu jádra procesoru. Samotné jádro procesoru ke své funkci vždy potřebuje periferie, ať už se jedná pouze o paměti, nebo i další bloky jako jsou čítače/časovače aj. Platforma v jazyce CodAL je tedy nadřazený modul samotného jádra procesoru. Na úrovni platformy je nutné instanciovat ASIP:

```
asip microblaze
{
    ia = "microblaze.ia";
    ca = "microblaze.ca_3stage";
};
```

Kromě samotného jádra je nutné také definovat paměti (příp. cache), sběrnice a porty a určit jejich vzájemné propojení. V následující ukázce je definovaná paměť programu o velikosti 4 KiB s rozhraním CLB. Tato paměť je přímo propojena se sběrnicí `ibus` definovanou v ASIPu. Kromě toho jsou definovány porty, které jsou opět přímo propojeny na rozhraní jádra procesoru.

```
// pamet programu
memory mem_program
{
    bits = { 32, 32, 8}; // sirka adresy/sirka slova/sirka LAU
    size = 0x1000;       // 4 KiB
    endianness = BIG;
    latencies = { 1, 1}; // latence - cteni/zapis

    interface read_only {
        type = CLB:SLAVE;
        flag = R;        // pouze pro cteni
    };
};

// porty platformy
port bit[1]  irq_in;  // port pro preruseni

// propojeni
connect microblaze.p_irq => irq_in;
connect microblaze.ibus  => mem_program.read_only;
```

Součástí procesorů jsou často i další periferie jako jsou bloky čítačů/časovačů, kontroler přerušení, složité paměťové systémy atd. Jazyk CodAL není určený pro popis takových bloků. Tyto bloky je vhodné popsat ve formě komponent (`component`). Na úrovni platformy je definováno pouze rozhraní komponenty a propojení s dalšími

bloky. Při generování simulátorů je pak automaticky vygenerována implementační obálka v jazyce C++, která může být doplněna a využita pro korektní simulaci komponenty. Pokud má být komponenta součástí HDL popisu platformy, musí uživatel doplnit také HDL popis komponenty.

V následující ukázce je definice komponenty jednoduchého časovače. Popis rozhraní je obdobný jako popis rozhraní procesoru. Komponentu lze připojit na CLB sběrnici a využít pro generování přerušení. Typ (`type`) slouží pro simulační účely.

```
// komponenta casovace
component timer0
{
    type = "timer_t";

    interface if_dbus {
        bits = { 32, 32, 8}; // sirka adresy/sirka slova/sirka LAU
        endianness = BIG;
        type = CLB:SLAVE;
        flag = RW;           // cteni i zapis
    };

    port bit[1] p_irq;       // interrupt request
};

// definice datove sbernice
bus dbus
{
    ...

    decoder = {
        // datova pamet
        0x00000000 .. 0x00001FFF : mem_data.data_rw,
        // casovac
        0x10000100 .. 0x100001FF : timer0.if_dbus,
    };
};

// generovani preruseni
connect microblaze.p_irq    => timer0.p_irq;
```

2 PRAKTICKÁ ČÁST

2.1 MicroBlaze

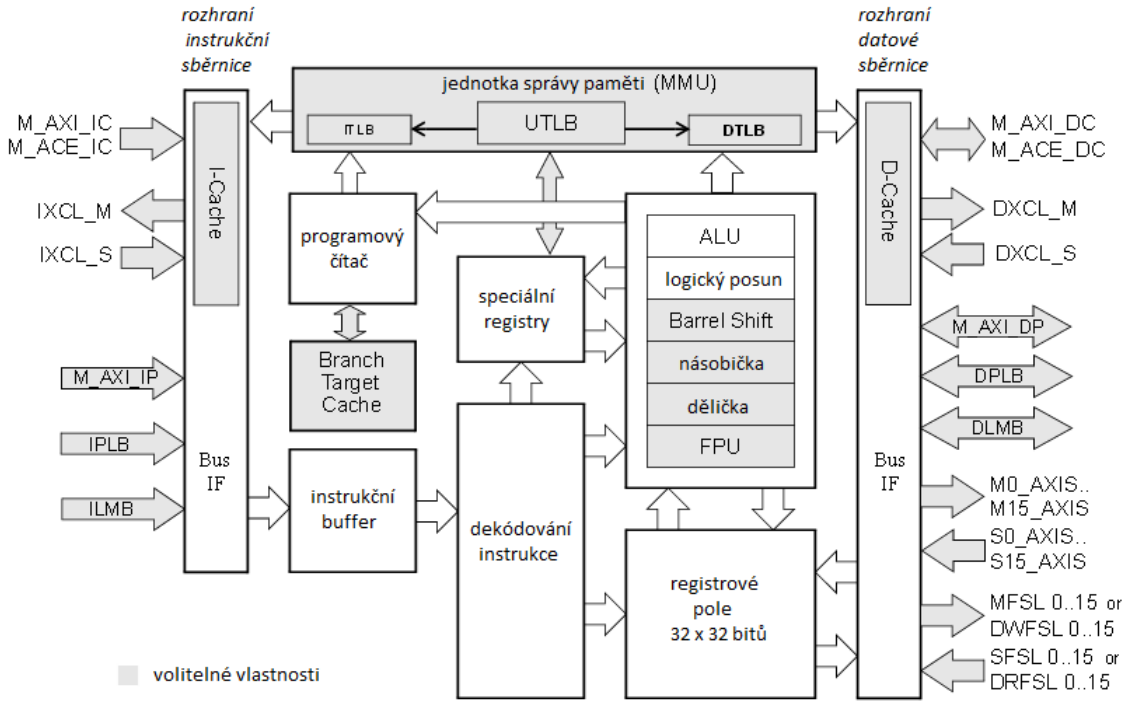
MicroBlaze je procesorové jádro určené na implementaci do programovatelných hradlových polí od firmy Xilinx. Jedná se o 32 bitový RISC procesor Harvardské architektury, který je možný v určitých parametrech přizpůsobovat aplikaci. Pevné parametry procesoru MicroBlaze jsou:

- registrové pole 32x32 bitů,
- 32 bitové instrukční slovo se 3 operandy a 2 adresnými módy,
- 32 bitová adresová sběrnice,
- jednoduchá zřetězená linka.

Kromě těchto parametrů je možné parametry před vygenerováním RTL popisu upravovat. Je tak možné upravovat konkrétní implementaci procesoru a to například v těchto směrech:

- počet stupňů linky (3 nebo 5),
- optimalizace na plochu či rychlost,
- instrukce pro logické posuny o více bitů (barrel shifter),
- hardwarová dělička a násobička,
- instrukce pro porovnání vzorů (pattern compare),
- implementace hardwarových výjimek,
- instrukční a datová vyrovnávací paměť (cache),
- optimalizace skoků (predikce),
- nastavení typu a parametrů sběrnice,
- ladící rozhraní.

Nastavením těchto parametrů lze nejen měnit mikroarchitekturu procesoru, ale přidávat a odebírat některé instrukce. Blokový diagram procesoru MicroBlaze je zobrazen na obrázku 2.1. Modrou barvou jsou naznačeny volitelné jednotky v architektuře procesoru - lze tedy přidat výpočetní jednotky jako jsou násobička, dělička, FPU (Floating Point Unit) a jednotka pro posuny o více bitů (barrel shifter). Je možné zvolit typ sběrnice (PLB - Processor Local Bus, LMB - Local Memory Bus, AXI - Advanced eXtensible Interface aj.) případně doplnit vyrovnávací paměti pro data a instrukce (cache). Také je možné přidat tzv. Branch Target Cache (BTC) pro predikci skoků a jednotku správy přístupu do paměti (MMU - Memory Management Unit) [14].



Obr. 2.1: Blokový diagram procesoru MicroBlaze [14]

Výrobce poskytuje přednastavené konfigurace [15], je možné zvolit například:

- minimální plocha (bez cache a ladícího rozhraní),
- maximální výkon (velké cache, ladící rozhraní, extra výpočetní jednotky),
- maximální frekvence (malé cache, bez debug a některých výpočetních jednotek),
- typický - kompromis mezi všemi parametry,
- Linux - vhodné nastavení pro běh operačního systému Linux (včetně MMU).

Firma Xilinx kromě samotné realizace procesoru dodává také překladač jazyka C. Překladač je možné nastavit podle mikroarchitektury procesoru nebo importovat nastavení pomocí nástrojů Xilinx. Překladač pak používá pouze instrukce, které jsou v procesoru implementovány.

2.1.1 Mikroarchitektura

MicroBlaze používá zřetězeného zpracování instrukcí. Počet stupňů linky má vliv jak na rychlost a výkon procesoru, tak na plochu. Každá instrukce prochází všemi stupni linky. Většina instrukcí je zpracována v každém stupni linky během jednoho hodinového taktu. Některé instrukce však potřebují na zpracování v EX (Execute) stupni taktů více - například instrukce s plovoucí řádovou čárkou. V ta-

kovém případě je zpracování instrukcí na několik taktů pozastaveno. V dokumentaci procesoru není zmíněno, zda hazardy v mikroarchitektuře řeší překladač nebo jsou řešeny hardwarově.

MicroBlaze je možné implementovat ve dvou verzích - se 3 nebo 5 stupni linky. Třístupňová linka je složena ze stupňů: IF (Instruction Fetch), ID (Instruction Decode) a EX (Execute). Pětistupňová linka rozšiřuje architekturu navíc o stupně MEM (Memory) a WB (Writeback).

U třístupňové architektury trvá zpracování některých instrukcí (násobení, posuny, ukládání a načítání z paměti, ...) více taktů. Tímto způsobem je ošetřena kritická logická cesta a procesor dosahuje vyššího pracovního kmitočtu, avšak za cenu zastavování linky. U načítání z paměti může být důvodem také synchronní přístup do paměti. U pětistupňové linky jsou tyto instrukce zpracovány ve více stupních linky, takže nemusí docházet k zastavování linky.

Skoky se vykonávají ze stupně EX. Trvají tedy 3 takty, z nichž 2 jsou potřeba pro znovunaplnění linky. Ke snížení penalizace jsou podporovány i skoky s jedním „delay“ slotem, penalizace je pak místo 3 taktů pouze 2 takty. Některé instrukce jsou však pro vkládání do „delay“ slotu zakázány (např. jiné skoky). Během vykonávání skoku s „delay“ slotem je také automaticky zakázáno přerušení.

2.1.2 Registry

MicroBlaze obsahuje 32 32-bitových registrů pro obecné použití. Toto registrové pole má 3 porty pro čtení a 1 port pro zápis. Všechny tyto registry jsou přímo přístupné instrukcemi. Některé registry jsou vyhrazeny pro speciální funkce:

- R0 - je vždy 0,
- R14 - ukládání návratové adresy pro přerušení,
- R15 - doporučeno pro ukládání návratové adresy funkce.

Dále obsahuje několik speciálních 32-bitových registrů. K těmto registrům je možné přistupovat instrukcemi `MFS` (Move From Special purpose register) a `MTS` (Move To Special purpose register). Registry, které budou implementovány v praktické části diplomové práce, jsou zobrazeny v tabulce 2.1. Ke každému registru je navíc uvedena jeho adresa a přístupnost registru přes instrukce `MFS` a `MTS`. Kromě těchto registrů procesor může obsahovat další registry, které slouží k obsluze výjimek (registry EAR, ESR, BTR, ...). Procesor MicroBlaze podporuje volitelnou implementaci různých výjimek (detekce neznámé instrukce, nezarovnaný přístup do paměti, kontrola přístupu na zásobník aj.). V rámci této práce budou běžné hardwarové výjimky implementované pouze jako kontrola při běhu simulátoru, v generovaném HDL popisu nebude přítomna logika těchto výjimek.

Tab. 2.1: Seznam implementovaných speciálních registrů

Speciální registr	**Adresa**	**Čtení (`MFS`)**	**Zápis (`MTS`)**
Program Counter (PC)	0x0	ANO	NE
Machine Status Register (MSR)	0x1	ANO	ANO
Processor Version Register (PVR)	0x2000	ANO	NE

PC (Program Counter)

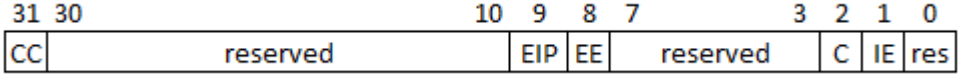
Programový čítač je 32-bitový registr, který určuje adresu aktuální instrukce. Při běžném vykonávání programu je PC vždy inkrementován o 4, v případě skoku je do registru PC uložena cílová adresa skoku. Protože MicroBlaze nepodporuje nezarovnaný přístup do paměti a všechny instrukce mají velikost 1 slova (32 bitů), spodní 2 bity jsou vždy nulové.

MSR (Machine Status Register)

Registr MSR (stavový registr) sdružuje všechny významné příznaky procesoru. Uspořádání příznaku v 32-bitovém registru MSR je zobrazeno na obrázku 2.2. Význam jednotlivých příznaků je:

- CC (Carry Copy) - kopie příznaku C (Carry), hodnota CC je při čtení vždy stejná jako C, zápis do bitu CC je ignorován,
- EIP (Exception In Progress) - příznak je nastaven pokud probíhá zpracování obsluhy výjimky,
- EE (Exception Enabled) - pokud je bit EE nastaven, pak je povoleno vyvolání obsluhy výjimky,
- C (Carry) - příznak přetečení aritmeticko-logických operací,
- IE (Interrupt Enable) - globální povolení přerušení.

Příznaky sloužící řízení výjimek (EE a EIP) nebudou implementovány v navrženém modelu, zápis do těchto příznaků bude ignorován a při čtení budou příslušné bity vždy v log. 0.



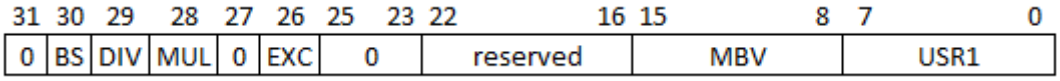
Obr. 2.2: Uspořádání registru MSR

PVR (Processor Version Register)

MicroBlaze obsahuje registr označovaný jako PVR - tento registr je určený pouze pro čtení a jeho hodnota je odvozena od implementované mikroarchitektury. Tímto způsobem lze ověřit verzi a implementované funkce procesoru.

Na obrázku 2.3 je zobrazeno uspořádání registru PVR. Význam jednotlivých bitů je:

- BS - 1 pokud je implementován Barrel Shifter,
- DIV - 1 pokud je implementována celočíselná dělička,
- MUL - 1 pokud je implementována celočíselná násobička,
- MBV (MicroBlaze release Version) - konstanta odpovídající verzi procesoru,
- USR1 - uživatelem volitelná konstanta.



Obr. 2.3: Uspořádání registru PVR

2.1.3 Reset, přerušení a výjimky

MicroBlaze podporuje reset, přerušení a několik druhů výjimek. Pro každou z těchto událostí jsou rezervovány adresy v paměti programu dle tabulky 2.2. U některých událostí navíc dojde také k uložení návratové hodnoty. Opět jsou uvedeny pouze události, které budou realizovány v praktické části práce.

Tab. 2.2: Rezervované adresy v paměti a registry pro uložení návratové hodnoty pro speciální události

Událost	**Adresa obsluhy**	**Registr pro uložení návratové hodnoty**
Reset	C_BASE_VECTORS + 0x00 - C_BASE_VECTORS + 0x04	-
Přerušení	C_BASE_VECTORS + 0x10 - C_BASE_VECTORS + 0x14	R14

Při resetu dochází k nastavení všech registrů v procesoru na výchozí hodnoty a vykonávání programu začíná od adresy C_BASE_VECTORS + 0x00. MicroBlaze

podporuje jeden externí zdroj přerušení na pinu „interrupt input port“, který je citlivý na úroveň signálu. Přerušení je možné vyvolat pouze tehdy, pokud je povoleno přerušení nastavením příznaku IE. V případě, že by byly implementovány i výjimky, nesmí být současně nastaven příznak EIP v registru MSR. Při vyvolání přerušení dojde k nahrazení instrukce v dekodéru instrukcí skoku na adresu obsluhy přerušení (C_BASE_VECTORS + 0x10), příznak IE je vynulován a adresa nahrazené instrukce je uložena do registru `R14`. Kvůli řešení hazardů v mikroarchitektuře nemusí být vyvolání přerušení okamžité, záleží na zpoždění pamětí a jiných faktorech. Důležité je, aby všechny předchozí instrukce byly dokončeny dříve, než dojde ke skoku na obsluhu přerušení. Při návratu z přerušení (instrukce `RTID`) je příznak IE opět nastaven. MicroBlaze podporuje i další módy přerušení - citlivost na hranu, přímé adresování více zdrojů přerušení. Tyto módy však nebudou v praktické části realizovány. Poslední ze speciálních událostí procesoru jsou hardwarové výjimky, které budou implementovány jako kontroly při běhu simulátoru.

2.2 Model na úrovni instrukcí

Model na instrukční úrovni specifikuje instrukční sadu procesoru, a to jak z pohledu binárního kódování, tak i funkce jednotlivých instrukcí. Jedná se o nejjednodušší model procesoru. Každá instrukce je modelována přesně podle své funkce na velice jednoduché úrovni, a to většinou bez potřeby pomocných signálů a registrů. Tím lze dosáhnout velké efektivity simulátoru. Díky jednoduchosti popisu lze celý model připravit a odladit v rámci několika dnů, maximálně týdnů.

Z modelu na instrukční úrovni se poté generují všechny SW nástroje (např. překladač jazyka C). Jedním z velmi užitečných nástrojů je profiler, jenž dokáže odhalit místa v programu, které zabírají nejdelší čas, analyzuje také použití instrukcí a vyhledává časté posloupnosti instrukcí. Lze pak snadno optimalizovat instrukční sadu pro konkrétní aplikaci a poměrně rychle ověřit její vliv na výsledcích simulace a to bez složitých zásahů do mikroarchitektury procesoru. Toto však není předmětem této práce, protože instrukční sada procesoru MicroBlaze je daná. Podrobné informace (včetně binárního kódování) lze nalézt v příručce procesoru MicroBlaze [14].

V rámci praktické práce byl vytvořen model procesoru MicroBlaze na instrukční úrovni. Některé instrukce jsou v modelu volitelné, jejich použití je možno povolit nastavením procesoru podobně jako je tomu u originální verze od výrobce. Tímto způsobem lze generovat programy, které jsou binárně kompatibilní s konkrétní vygenerovanou verzí procesoru MicroBlaze. Do modelu byla implementována obsluha přerušení a detekce výjimek v simulátoru (např. nezarovnaný přístup do paměti). Celý model byl doplněn o volitelné výpisy, které lze využít při ladění programů.

2.2.1 Instrukční sada

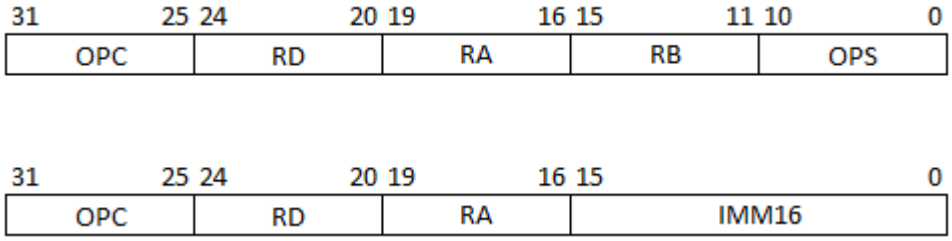
Pro model bylo z instrukční sady procesoru MicroBlaze vybráno celkem 140 instrukcí, které odpovídají základní instrukční sadě procesoru s těmito rozšířeními:

- instrukce pro logické posuny o více bitů (barrel shifter),
- hardwarová násobička,
- instrukce pro porovnání vzorů (pattern compare),
- instrukce pro nastavení a nulování registru MSR (`MSRSET` a `MSRCLR`).

Některé další instrukce nebyly implementovány, a to zejména z těchto důvodů:

- zbytečně velké hardwarové nároky (instrukce pro dělení),
- neimplementované funkce či bloky (instrukce pro cache paměti, FPU instrukce, návrat z obsluhy výjimek `RTED` a další).

Instrukce lze podle instrukčního formátu (2.4) rozdělit na 2 základní skupiny: instrukce registrové a instrukce využívající konstantu. Pozice indexů registrového pole, 16-bitové konstanty a hlavního operačního kódu (OPC) je v binárním kódování fixní. Hlavní operační kód slouží k rozlišení skupin instrukcí, další dělení pomocí vedlejších operačních kódů již není v instrukcích fixní a využívá zbývající části kódu instrukce (např. OPS pro tříregistrové instrukce; RD pro instrukce, které nezapisují do registrového pole atd.). Kompletní seznam implementovaných instrukcí lze nalézt v příloze v tabulce A.1.



Obr. 2.4: Instrukční formáty procesoru MicroBlaze

U všech instrukcí bylo snahou dodržet přesnou kompatibilitu s originálním jádrem firmy Xilinx. Jedinou výjimkou jsou instrukce pro odčítání s carry (`subc`, `sub`, `subkc`). Tyto instrukce byly implementovány jako odčítání s borrow, protože použitá verze nástrojů odčítání s carry nepodporuje. Podrobnosti o rozdílu těchto dvou typů odčítání lze nalézt v [16].

Model byl doplněn o 3 další instrukce, konkrétně se jedná o instrukce `HALT`, `SYSCALL` a `CALL_INT`. Instrukce `HALT` a `SYSCALL` slouží pouze pro simulaci a v generovaném RTL popisu se chovají jako `NOP`. Instrukce `HALT` slouží jako detekce konce

programu a to jak v simulátorech tak při funkční verifikaci. Instrukce `SYSCALL` (systémové volání) je základem pro implementaci knihovny, která obsahuje mechanismus volání funkce operačního systému. V programech v jazyce C je poté možné používat standardní systémové funkce (tisk do konzole, práce se soubory aj.). Tyto funkce mohou usnadnit ladění a testování programů i samotného procesoru, avšak pouze při simulacích.

Instrukce `CALL_INT` je instrukce, která je vložena do dekodéru v případě přerušení. Jde o skok na absolutní adresu obsluhy přerušení, který současně zakáže další přerušení vynulováním příznaku IE ve stavovém registru. Pro návrat z obsluhy přerušení se na rozdíl od ostatních instrukcí volání podprogramu (`BRALID`, `BRLID`, `BRALD` a `BRLD`) ukládá přímo adresa instrukce, protože při vložení do dekodéru je aktuálně načtená instrukce programu nahrazena instrukcí `CALL_INT`. Při návratu z přerušení tedy musí být znovu vykonána i původně nahrazená instrukce. Instrukce `CALL_INT` není překladačem využívána, v principu ji však lze využít i pro softwarové volání přerušení. V takovém případě je však nutné upravit obsluhu přerušení tak, aby návrat probíhal na inkrementovanou návratovou adresu.

Každá z instrukcí, která používá konstantu (ve svém názvu má na konci `I` jako immediate), má ve svém binárním kódování 16-ti bitovou konstantu, která je automaticky znaménkově rozšířena na 32 bitů. V některých případech však nemusí 16-ti bitová konstanta stačit, v takových případech může každá z těchto instrukcí využít rozšíření pomocí jiné konstanty, pokud tuto instrukci předchází instrukce `IMM`. Instrukce v tomto případě nepoužije znaménkové rozšíření, ale na pozici významnějších bitů je vložena přímo konstanta zakódovaná u instrukce `IMM`.

```
i1: IMM 0x1234
i2: ADDI R10, R5, 0x5678     // R5 + 0x12345678
```

Instrukční sada procesoru MicroBlaze nemá vyhrazenou `NOP` instrukci, jako náhradu využívá instrukci `ADD R0, R0, R0`. Protože je registr `R0` vždy 0, tato instrukce v procesoru nevyvolá žádnou změnu a v principu se jedná o `NOP` instrukci.

2.2.2 Generování překladače

Z modelu na instrukční úrovni je možné vygenerovat překladač, který je postaven na LLVM (Low Level Virtual Machine). LLVM je komplexní překladový systém, jenž poskytuje nástroje pro tvorbu vlastního překladače. Skládá se ze 3 částí: „front-end“, optimalizér a „back-end“. Pouze „front-end“ je jazykově závislý, slouží k analýze programu a překladu do tzv. LLVM IR kódu, který je určen pro další zpracování. Pro překlad jazyka C (také C++) se používá „front-end“ clang. Soubor knihoven

LLVM Core obsahuje optimalizér a „back-end“ překladače a slouží k překladu LLVM IR kódu na konkrétní program v jazyce assembler [17].

Každá instrukce, jejíž zpracování trvá více taktů, byla v modelu doplněna o tzv. `schedule class`. Tyto třídy slouží zejména k definici latence instrukce, tedy za kolik taktů je uložen výsledek instrukce. Tato informace je pak použita překladačem při plánování instrukcí a přeskládání instrukcí za účelem odstranění RAW hazardů. Pro neodstraněné RAW hazardy může překladač vložit `NOP` instrukce, avšak tato volba je v modelu vypnuta a RAW hazardy jsou detekovány a ošetřeny přímo v mikroarchitektuře. Další využití tříd může být k definici počtu „delay“ slotů skoku nebo pro zakázání plnění „delay“ slotů některými instrukcemi.

```
// definice schedule class
schedule_class loads
{
    // latence 3 takty
    latency = 3;
};

// pouziti schedule class v semantice instrukce
codasip_compiler_schedule_class(loads);
```

Při generování překladače je z instrukcí extrahována sémantika, která je mapována na konkrétní operace LLVM. Tyto instrukce modelu jsou pak používány při překladu LLVM operací na instrukce procesoru. Při generování překladače je nutné nalézt alespoň základní sadu instrukcí – typicky se jedná o aritmetické operace, čtení a zápis do paměti, načítání konstanty do registru, podmíněné skoky na základě porovnání hodnot uložených ve dvou registrech a další. V případě, že některé z těchto instrukcí nebyly nalezeny, je potřeba takové instrukce doplnit nebo překladači poskytnout popis, jak tyto instrukce nahradit.

Nejjednodušší formou, jak doplnit chybějící instrukce, je definice uživatelských pravidel překladače (některé obecné základní pravidla jsou již definovány). Tímto způsobem lze nahradit některé jednoduché operace, obvykle komplementární operací. V následujícím příkladu je definováno pravidlo, které říká, že instrukce porovnání dvou registrů (a větší než b) lze nahradit provedením opačné logické operace s přehozeným pořadím registrů (b menší než a).

```
// pravidlo pro instrukci porovnani hodnot 2 registru
(setgt reg:0, op:1):    // (a>b)
    (setlt :1, :0);     // (b<a)
```

Složitější případy je nutné řešit pomocí tzv. „compiler alias“ (alias pro překladač) nebo tzv. „dummy“ instrukcí. Oba tyto přístupy jsou velice podobné, avšak mají

své výhody a nevýhody. Lze jimi definovat posloupnosti instrukcí, které společně mohou vytvořit překladačem vyžadovanou funkci.

Jednoduchým příkladem, kde je výhodné využít alias pro překladač, může být načítání 32-bitové konstanty do registru. Alias pro překladač vytváří náhradní sémantiku již existující instrukce a to pouze pro překladač. V tomto konkrétním případě je vytvořena instrukce pro překladač o dvojnásobné délce (64 bitů), protože jde o spojení 2 instrukcí - instrukcí `IMM` a `ORI`. Překladač alias používá jako jednu instrukci, při běhu procesoru jsou však obě instrukce vykonány samostatně (dle původní sémantiky). Je však nutné zakázat vkládání této skupiny instrukcí do „delay“ slotů skoků pomocí třídy `no_delay_slot`.

```
// ukladani 32-bitove konstanty do registru - alias pro prekladac
element i_movi32_calias : compiler_alias(i_imm, i_logical_imm)
{
    use gpreg as reg_dst;
    use simm32;

    assembler {
        "IMM" simm32 ">> 16"
        "ORI" reg_dst "," "R0" "," simm32
    };

    ...

    semantics
    {
        // zakaze vkladani do delay slotu
        codasip_compiler_schedule_class(no_delay_slot);

        // semantika pro prekladac
        rf_gpr[reg_dst] = simm32;
    };
};
```

U „dummy“ instrukce naopak překladač vidí, že jde o spojení více instrukcí. Dokáže je tedy optimálně využívat pro řešení hazardů. Na konkrétním příkladě je zobrazena „dummy“ instrukce, která emuluje podmíněný relativní skok na základě porovnání dvou registrů, v tomto případě znaménkové porovnání RA je větší než RB. U „dummy“ instrukcí se opět uvádí sémantika, místo zápisu v jazyce assembler se však uvádí skupina, do které instrukce patří, spolu s operandy, které používá. Přesný popis použití a původ názvů a klíčových slov lze nalézt v [11]. Tento zápis je uživatelsky poměrně nepřívětivý, ale umožňuje vytvořit instrukce i nad rámec možností jazyka CodAL.

```
// dummy instrukce pro podmineny relativni skok
dummy_instr brcond_imm__setgt,
   { _reg_2 = regop(_reg), _reg_1 = regop(_reg), imm_1 = immop() },
    // nacteni registrovych operandu
    %7a = i32 regop(_reg_1);;
    %6a = i32 regop(_reg_2);;

    // porovnani regA > regB
    %5a = setgt(%7a,%6a);

    // nacteni aktualni hodnoty PC
    %8b = i32 getcurrpc();

    // nacteni offsetu relativniho skoku - znamenkove rozsireni
    %7b = i16 immop(imm_1,1,0);
    %6b = i32 sign_extend(%7b);

    // vypocet cile skoku
    %5b = add(%6b,%8b);
    %9b = i32 -4;
    %4b = add(%5b,%9b);

    // podmineny skok
    brc(%4b,%5a);

    ,
    // instrukce, ze kterych je emulace slozena
    i_compare__ops_cmp___reg___reg___reg__ %tmp, _reg_1, _reg_2;
    i_cond_br_imm__ops_bgei___reg__rel_addr16__ %tmp, imm_1;
```

Pro překladač je také nutné specifikovat ABI (Application Binary Interface). Jde o nízkoúrovňové rozhraní obsahující soubor pravidel na úrovni překladače. ABI definuje např. význam jednotlivých registrů (které registry jsou využity pro předávání parametrů funkcí, rezervované registry, ukazatel na zásobník atd.), validní datové typy pro překladač ale i např. směr zásobníku [18].

Poslední důležitou částí, kterou je potřebné nadefinovat, je „startup“ kód označovaný jako crt0. Jedná se o program v assembleru, jenž slouží k inicializaci procesoru po resetu a volání funkce `main`, která je povinnou funkcí všech programů v jazyce C. V následující ukázce je jednoduchý příklad, který inicializuje ukazatel na zásobník SP (Stack Pointer, registr R1) a base pointer (registr R2). Před voláním funkce `main` uloží do registru návratové hodnoty (R3) 1. Při správném vykonání funkce by měla být návratová hodnota programem přepsána na 0. Po návratu z funkce `main` je procesor zastaven pomocí instrukce `HALT`.

```
// tato sekce je vzdy na adrese 0.
.section .crt0, "x"
_start:
.global _start

// nacteni stack pointeru
IMM _stack >> 16
ORI R1, R0, _stack & 0xffff

// nacteni base pointeru
OR R2, R0, R1

// prepsani navratove hodnoty na 1
ORI R3, R0, 1

// volani funkce main
IMM $main >> 16
BRALID R15, $main & 0xffff
NOP

// zastaveni procesoru po navratu z funkce main
HALT
```

Generovaný překladač dokáže pouze omezeně využívat rozšíření konstanty instrukce pomocí předcházející instrukce `IMM`. Pro lepší výkon by byly třeba ruční zásahy a optimalizace v nástrojích. Protože se však nejedná o komerčně využitelné jádro, nejsou tyto zásahy prioritou Codasip týmu a bude využíván běžně generovaný překladač bez úprav.

2.3 Model na RTL úrovni

Model na RTL úrovni popisuje mikroarchitekturu procesoru. Oproti modelu na instrukční úrovni je tento model mnohem složitější, protože respektuje časování na hardwarové úrovni. Vzhledem k větší složitosti je obvykle navrhován až po vytvoření kompletní instrukční sady procesoru, aby v pozdější fázi návrhu nebyly nutné zásahy do navržené mikroarchitektury.

Z modelu na RTL úrovni je možné vygenerovat HDL popis procesoru v jazyce VHDL, Verilog nebo SystemC, který je základem pro implementaci do obvodu FPGA nebo výrobu zákaznických integrovaných obvodů ASIC (Application-Specific Integrated Circuit). Model na RTL úrovni však slouží také ke generování CA simulátoru. Tento simulátor je mnohonásobně rychlejší než simulace na úrovni HDL jazyků. Velkou výhodou je také možnost implementace pomocných výpisů, které

mohou pomoci odhalit chyby v návrhu při běhu simulátoru. Pro ladění chyb je také možné krokovat běh procesoru přímo v jazyce CodAL.

V rámci praktické práce byl vytvořen model procesoru MicroBlaze na RTL úrovni a to ve 2 verzích. Obě verze jsou kompatibilní s modelem na instrukční úrovni, navzájem se liší pouze počtem stupňů mikroarchitektury. Stejně jako u originálního jádra MicroBlaze byla implementována 3 stupňová mikroarchitektura optimalizovaná na plochu a 5 stupňová mikroarchitektura optimalizovaná na výpočetní výkon. Některé hardwarové bloky jsou v CA modelech implementovány volitelně tak, že respektují aktuální nastavení instrukční sady procesoru. Oba modely byly také doplněny o volitelné výpisy, které lze využít při ladění.

Z důvodu ochrany firmou utajených postupů optimalizace mikroarchitektur procesorů jsou na přiloženém CD dostupné pouze neoptimalizované verze navržených modelů.

2.3.1 Třístupňová mikroarchitektura

První z vytvořených CA modelů implementuje 3 stupňovou mikroarchitekturu procesoru, jejíž zjednodušené schéma je zobrazeno v příloze na obrázku B.1. Na obrázku jsou zobrazeny pouze datové cesty; řídící signály a logika řídící běh linky nejsou pro přehlednost zobrazeny.

Paměti jsou v modelu připojeny přes rozhraní sběrnice CLB (Codasip Local Bus). Programová i datová paměť jsou připojeny přes oddělené sběrnice tak, aby nedocházelo ke strukturálním hazardům. Sběrnice CLB je synchronní sběrnicí, zápis i čtení dat přes sběrnici trvá za běžných okolností 2 takty (např. při připojení pamětí cache může být zpoždění vyšší). Zjednodušeně lze komunikaci přes CLB sběrnici popsat následujícím způsobem: v prvním taktu je společně s adresou vydán požadavek (v případě MicroBlaze pouze čtení nebo zápis), na který sběrnice vrací svůj status - zjednodušeně se dá říci, že vrací odpověď, zda byl požadavek přijat nebo musí být zopakován. V případě přijetí požadavku jsou následující hodinový takt přeneseny data (od procesoru v případě zápisu, od paměti v případě čtení). Současně však sběrnice vrací i odpověď (response), zda přístup proběhl v pořádku, je třeba počkat, nebo proběhl s chybou (nezarovnaný přístup do paměti, přístup mimo paměť atd.). V případě čekání musí být celá druhá část komunikace zopakována [12].

Linka je složena ze tří stupňů:

1. FE (Fetch) – načítání instrukcí z programové paměti,
2. ID (Instruction Decode) – dekódování instrukce a čtení instrukčních operandů,
3. EX (Execute) – vykonání instrukcí, přístup do datové paměti a zápis do registrového pole.

Jednotlivé stupně jsou odděleny registry. Mezi stupni FE a ID plní funkci oddělovacího registru sběrnice CLB. Každá instrukce prochází všemi stupni linky, prakticky tedy trvá zpracování každé instrukce minimálně 3 hodinové takty. V praxi se jako trvání zpracování instrukce většinou uvádí počet taktů, kolik trvá samotný výpočet (v tomto případě v EX stupni linky). U většiny instrukcí je výsledek vypočten během jednoho hodinového taktu, některé instrukce však mohou trvat déle. Důvodem může být snaha o zvýšení maximální pracovní frekvence rozdělením složitého výpočtu přes několik registrů (např. u násobení) nebo vlastnosti sběrnice při přístupu do paměti (obvykle synchronní přístup). Linku je třeba řídit s ohledem na instrukce, jejichž zpracování trvá více hodinových taktů. Při zpracovávání těchto instrukcí je nutné linku zastavit na počet taktů potřebný k dokončení operace.

Stupeň FE (Fetch)

Stupeň FE je prvním stupněm ve zpracování instrukce. V tomto stupni je implementován programový čítač PC a logika zajišťující výpočet adresy následující instrukce.

Hodnota programového čítače je adresa pro čtení z programové paměti. Současně s požadavkem do programové paměti je vypočtena i adresa následující instrukce. Pokud nedochází ke skoku, je hodnota programového čítače inkrementována o 4, při vykonávání skoků je do PC uložena adresa cíle skoku z EX stupně linky. V případě zastavení linky je nutné pozdržet i inkrementaci PC a případně zopakovat požadavek do paměti tak, aby nedošlo ke ztrátě instrukce. Ze stupně FE může dojít k zastavení linky pouze v případě, že požadavek na čtení z paměti programu není přijat.

Stupeň ID (Instruction Decode)

Ve stupni ID je dokončeno čtení instrukce z instrukční paměti. V případě přerušení je načtené instrukční slovo nahrazeno instrukcí `CALL_INT`. Nahrazení instrukce je však možné pouze pokud nejde o instrukci v „delay“ slotu skoku, je povoleno přerušení ve stavovém registru a pokud předchozí instrukce nebyla instrukce `IMM`. Instrukce `IMM` tvoří s následující instrukcí dvojici, která je vykonávána společně a nelze ji rozdělit. Vložení instrukce `CALL_INT` je v takovém případě zpožděno až do vykonání celé dvojice instrukcí.

Z instrukčního slova jsou vybrány bity určené k dekódování instrukce. V dekodéru instrukce jsou pak nastaveny všechny řídící signály. Ve stupni ID jsou také vybrány operandy instrukce. 16-bitová konstanta je rozšířena buď znaménkově nebo sloučením s uloženou 16-bitovou konstantou z předchozí instrukce `IMM`. Adresy pro přístup do registrového pole jsou zakódovány přímo v binárním kódování instrukce. Při čtení

z registrového pole je implementováno přeposílání dat (forwarding), takže v lince nevznikají RAW hazardy. Tato logika však kvůli přehlednosti není na obrázku B.1 naznačena. Přeposílání dat je implementováno jako statická kontrola adresy pro čtení v ID stupni s adresou pro zápis ve stupni EX. Pokud se adresy shodují a současně je ve stupni EX povolen zápis do registrového pole, jsou místo dat z registrového pole předána zapisovaná data ze stupně EX. Výhodou 3 stupňové linky je, že přeposílání probíhá vždy ze stupně EX, a to i při zpracování instrukcí, které trvají více taktů. Díky implementaci přeposílání v lince nenastávají RAW hazardy. Při přeposílání není potřeba kontrolovat, jestli je výsledek už platný, protože řízení linky nedovolí přechod instrukce do dalšího stupně před dokončením předchozí instrukce.

Ze stupně ID může dojít k zastavení linky, pokud operace čtení z programové paměti není dokončena. Na konci stupně jsou všechny informace potřebné pro vykonání instrukce (operandy i řídící signály) uloženy do registrů.

Stupeň EX (Execute)

Ve stupni EX jsou vykonány jednotlivé instrukce. Jak je naznačeno na obrázku B.1, zpracování některých instrukcí trvá více hodinových taktů. Příkladem mohou být instrukce pro násobení, které trvají 3 hodinové takty. Přístup do paměti přes CLB sběrnici trvá 2 a více taktů, podle stavu sběrnice. Celá linka musí být při zpracování těchto instrukcí zastavena do doby, dokud není instrukce ve stupni EX zpracována.

Celý EX stupeň je rozdělen na několik funkčních jednotek, z nichž některé jsou implementovány volitelně. Ukládá-li aktuálně zpracovávaná instrukce výsledek do registrového pole, je vybrán výsledek z jedné z funkčních jednotek podle řídícího signálu. Samotný zápis je povolen až v posledním taktu zpracování instrukce, zejména proto, aby se v modelu negenerovaly falešné zápisy do registrového pole.

V lince se skoky vyhodnocují také ze stupně EX. Při výpočtu adresy relativních skoků je využívána sčítačka v aritmeticko-logické jednotce. Pokud se skáče, je nutné, aby řízení linky zabezpečilo vymazání načtených instrukcí v lince s ohledem na „delay“ sloty instrukce skoku.

2.3.2 Pětistupňová mikroarchitektura

Druhý z vytvořených CA modelů implementuje 5 stupňovou mikroarchitekturu procesoru, jejíž zjednodušené schéma je zobrazeno v příloze na obrázku B.2. Třístupňová linka byla rozšířena o další 2 stupně, které znamenají zejména vyšší počet oddělovacích registrů. Na obrázku jsou z důvodu přehlednosti opět zobrazeny pouze datové cesty. Stejně jako u třístupňové linky jsou paměti připojeny přes oddělené rozhraní sběrnice CLB.

Linka je složena z pěti stupňů:

1. FE (Fetch) – načítání instrukcí z programové paměti,
2. ID (Instruction Decode) – dekódování instrukce a čtení instrukčních operandů,
3. EX (Execute) – vykonání instrukcí, výpočet adresy pro přístup do datové paměti,
4. MEM (Memory access) – přístup do datové paměti,
5. WB (Writeback) – zápis do registrového pole.

Každá instrukce opět prochází všemi stupni linky, prakticky tedy trvá zpracování každé instrukce minimálně 5 taktů. Výhodou delší linky je, že instrukce, jejichž zpracování trvá více hodinových taktů, nemusí zastavovat linku, ale jejich výpočet probíhá přes několik stupňů linky. Prakticky se tedy eliminuje zastavování linky z důvodu zřetězeného výpočtu násobení, synchronního přístupu do paměti atd. Další výhodou může být rozdělení kritické cesty v mikroarchitektuře a zvýšení maximálního pracovního kmitočtu.

Nevýhodou pětistupňové linky je složitější implementace přeposílání dat a nutnost detekce hazardů. Při přeposílání je třeba kontrolovat více stupňů současně. Navíc je nutné kontrolovat, zda nedochází k hazardům vlivem přeposlání výsledku operace, která ještě není dokončena. U zřetězených výpočtů je možné přeposílat až výsledek operace, který je dostupný v pozdějších stupních linky.

Stupeň FE (Fetch)

Stupeň FE je prvním stupněm ve zpracování instrukce. V tomto stupni je implementován programový čítač PC a logika zajišťující výpočet adresy následující instrukce. Implementace je shodná s implementací u třístupňové mikroarchitektury.

Stupeň ID (Instruction Decode)

Také stupeň ID je implementován podobně jako u 3 stupňové verze. Opět dochází k dokončení čtení instrukce z instrukční paměti, případně nahrazení instrukčního slova instrukcí `CALL_INT`. Instrukce je následně dekódována a jsou nastaveny všechny řídící signály. Vybírání operandů a rozšíření 16-bitové konstanty je shodné s třístupňovou verzí. Jediným rozdílem mezi jednotlivými verzemi je implementace přeposílání a detekce hazardů v lince. Tato logika opět není kvůli přehlednosti zobrazena na obrázku B.2.

Přeposílání dat je implementováno jako statická kontrola adresy pro čtení v ID stupni s adresou pro zápis ve stupni EX, MEM i WB. Pokud se adresa pro čtení shoduje s jednou z adres v následujících stupních, jsou daná data přeposlány. Priorita přeposílání klesá ve směru linky, tedy nejvyšší prioritu má následující stupeň EX,

protože se v něm nachází předchozí, poslední zpracovaná instrukce. Pro přeposílání musí být v daném stupni linky povolen zápis do registrového pole.

Protože zpracování některých instrukcí trvá více taktů bez zastavení linky, může se stát, že výsledek není připraven k přeposlání. Příkladem může být instrukce násobení, jejíž výsledek je připraven až ve stupni WB a v předcházejících stupních jej nelze přeposlat. K ošetření těchto hazardů může být využit překladač, který přeskládá instrukce podle latence. Hazardy neodstraněné přeskládáním instrukcí poté může odstranit vložením instrukcí `NOP`. Generovaný překladač byl nastaven tak, aby přeskládával instrukce podle latence, avšak nevkládal instrukce NOP.

Hazardy tedy mohou nastat a jejich detekce je implementována hardwarově. V případě detekce je nutné zastavit linku až do dokončení a přeposlání výsledku. Při zastavení linky se do následujícího stupně hardwarově vkládá instrukce `NOP`. Kromě kontroly adres a povolení zápisu instrukce je tedy potřeba kontrolovat, zda lze výsledek z daného stupně přeposlat. K efektivnímu využití linky se současně kontroluje, zda instrukce v dekodéru používá data z registru, který způsobuje hazard. V některých situacích se může stát, že ačkoli se adresy shodují, instrukce hodnotu registru nevyužívá a linka by byla zastavena zbytečně.

Stupeň EX (Execute)

Ve stupni EX jsou vykonány jednotlivé instrukce. Více taktové funkční jednotky jsou implementovány přes více stupňů, takže nedochází k zastavování linky. Některé funkční jednotky jsou opět implementovány volitelně podle aktuálního nastavení instrukční sady. U instrukcí, které jsou dokončeny během jednoho taktu přímo ve stupni EX, je vybrán výsledek z jedné z funkčních jednotek určený pro přeposlání.

Logika vyhodnocující skoky je implementována stejně jako u třístupňové linky. Pro výpočet adresy skoku se opět využívá sčítačka v aritmeticko-logické jednotce. Při skoku jsou vymazány instrukce z linky s ohledem na jeden „delay“ slot skoku.

Stupeň MEM (Memory access)

Ve stupni MEM je u instrukcí `LD` a `ST` vydáván požadavek do datové paměti. V případě, že sběrnice není připravena přijmout požadavek, je linka zastavena. Ve stupni MEM je také dokončen výpočet v jednotce „barrel shifter“ a výsledek je případně vybrán k přeposlání do stupně ID.

Stupeň WB (Writeback)

Ve stupni WB je u instrukcí `LD` a `ST` dokončen přístup do datové paměti. Současně je dokončen výsledek třítaktového násobení. Výsledek instrukce je vybrán pro pře-

poslání a zapsán do registrového pole. Zápisem do registrového pole pouze ze stupně WB se předchází vzniku strukturálních hazardů.

U zřetězené násobičky nejsou registry mezi stupni linky řízeny. Výpočet tedy probíhá nezávisle na běhu linky. Pokud během výpočtu došlo k zastavení linky, dojde k situaci, kdy výsledek násobení je ve stupni WB současně s předcházející instrukcí. Během výpočtu násobení se tedy detekuje zastavení linky a výsledek násobení je případně zazálohován do pomocného registru. Samotná instrukce násobení pak ve stupni WB v této situaci nepoužívá výsledek ze zřetězené násobičky, ale zálohu výsledku v pomocném registru. Tento pomocný registr není kvůli přehlednosti zobrazen na obrázku B.2.

2.3.3 Porovnání výsledků syntézy

Pro porovnání s jádrem od výrobce byla vytvořena jednoduchá platforma obsahující jádro procesoru se 2 oddělenými pamětmi (program a data) o velikosti 2 KiB. Syntéza byla prováděna v nástroji Xilinx ISE Design Suite 14.7 pro FPGA Artix7 XC7A100T-1CSG324. Zdrojové kódy použité při srovnání, stejně jako optimalizovaná verze modelu, nejsou z důvodu utajení součástí přiloženého CD.

Srovnání výsledků syntézy třístupňové verze procesoru je zobrazeno v tabulce 2.3, pro syntézu byla zvolena optimalizace na plochu. Pětistupňová verze procesoru byla syntetizována s optimalizací na rychlost, srovnání výsledků je zobrazeno v tabulce 2.4. Pro porovnání bylo použito jádro MicroBlaze verze 8.50.

Tab. 2.3: Porovnání výsledků syntézy pro 3 stupňovou mikroarchitekturu

Logické bloky	**Xilinx implementace**	**Vlastní implementace**	**Poměr**
Registry	888	382	43,0 %
LUT	970	1268	130,7 %
BRAM	4	4	100 %
DSP	4	4	100 %
Mezní frekvence	159,6 MHz	114,0 MHz	71,4 %

Při syntéze z vygenerovaných HDL zdrojových kódů se uplatňuje určitá variabilita výsledků. Výsledky lze částečně ovlivnit nastavením nástroje Xilinx ISE Design Suite. Dle očekávání vlastní návrh nedosahuje takové optimalizace jako komerčně dostupné procesorové jádro. Na rozdíl od komerčního jádra je však možné navrženou mikroarchitekturu přizpůsobit vlastním požadavkům. Pro dosažení lepších výsledků je jedna z možností například úprava počtů stupňů linky.

Tab. 2.4: Porovnání výsledků syntézy pro 5 stupňovou mikroarchitekturu

Logické bloky	**Xilinx implementace**	**Vlastní implementace**	**Poměr**
Registry	1230	675	54,9 %
LUT	1317	1938	147,2 %
BRAM	4	4	100 %
DSP	4	4	100 %
Mezní frekvence	178,2 MHz	135,6 MHz	76,1 %

2.4 Testování a verifikace

Návrh procesorů je složitý proces, jehož správnost je před uvedením do funkce nutné co nejvíce ověřit. K tomuto účelu je v Codasip Studiu integrováno několik funkcí pro usnadnění testování a verifikace. Zvláštní význam verifikace nabývá u procesorů a systémů, které jsou určeny pro výrobu zákaznických integrovaných obvodů ASIC. Tento proces je finančně velmi náročný, je tedy snahou odhalit všechny chyby již během návrhu tak, aby i první vzorky byly funkční.

2.4.1 Testování pomocí sady testovacích programů

Funkce procesoru byla nejprve otestována pomocí testovací sady 703 programů a to ve 4 úrovních optimalizace překladače. Jedná se o testování na úrovni simulátorů. Po běhu každého programu se vyhodnocuje návratová hodnota programu uložená v registru. Konec programu je vyhodnocován vykonáním instrukce `HALT`. Pokud program skončí s návratovou hodnotou 0, pak je test vyhodnocen jako správný („OK“). Jiná návratová hodnota znamená chybu, která je označena jako „EC“ (Exit Code). Tato chyba může být způsobena jak špatným překladem tak i chybou v modelu. U některých testů se může stát (obvykle u modelu na RTL úrovni nebo složitých modelů), že test není dokončen během daného časového limitu, tento test je označen jako „TO“ (Time-Out). Řešením může být zvýšení časového limitu pro běh testu. Výsledky běhu simulátorů na testovací sadě jsou zobrazeny v tabulce 2.5.

Při běhu simulátoru byl nastaven časový limit 1 minuta pro IA model, 10 minut pro CA model. Z celé sady neprochází pouze 2 testy na optimalizaci `-O2`. Laděním těchto testů bylo zjištěno, že jde o chyby při překladu programu. Je nutné poznamenat, že tento typ testování slouží pouze k rychlému odhalení zásadních chyb. To je dané způsobem vyhodnocení správnosti testů – tedy jediné hodnoty na konci programu.

Tab. 2.5: Výsledek simulace testovací sady

Optimalizace	Model	OK	TO	EC
-O0 (bez optimalizace)	IA	703	0	0
	CA 3 st.	703	0	0
	CA 5 st.	703	0	0
-O1	IA	701	1	1
	CA 3 st.	701	1	1
	CA 5 st.	701	1	1
-O2	IA	703	0	0
	CA 3 st.	703	0	0
	CA 5 st.	703	0	0
-O3 (nejvyšší optimalizace)	IA	703	0	0
	CA 3 st.	703	0	0
	CA 5 st.	703	0	0

Další úrovní testování je porovnávání shody zápisů do architekturálních registrů (registrového pole) a transakcí přes sběrnice mezi IA a CA modelem. Tímto způsobem lze odhalit chyby, které se mohou projevovat, i když výsledek programu samotného je v obou případech správný.

2.4.2 Testy výkonnosti

Kromě testovacích programů byl procesor otestován pomocí sady testů výkonnosti označované jako „benchmarks“. Jedná se o sadů speciálních testů, které jsou určeny k porovnání procesorů napříč architekturami. Tyto testy jsou primárně určeny k snadnému porovnání výkonu verzí jednoho procesoru (např. po optimalizaci instrukční sady), ale i procesorů navzájem odlišných. S výhodou lze tyto testy využít i jako složitější testovací sadu.

Výsledek testu Dhrystone [19] je zobrazen v tabulce 2.6. Lze předpokládat, že v tomto případě jde přímo o porovnání výkonu generovaného překladače oproti originálnímu překladači, protože mikroarchitektury jsou z pohledu časování shodné a jediný rozdíl je tedy v překladači. U hodnot uváděných výrobcem však nelze určit přesné nastavení jádra i jediná instrukce navíc pak může výrazně měnit výsledky. Rozdíl může být způsoben i známými omezeními v generovaném překladači, které by mohly být odstraněny při dalších optimalizacích nástrojů. Ačkoli generovaný překladač dosahuje menšího výkonu, z výsledků je pozorovatelný nárůst výkonu pro pětistupňovou mikroarchitekturu.

Tab. 2.6: Porovnání výsledků testu Dhrystone

Implementace	**3 stupňová mikroarchitektura [DMIPS/MHz]**	**5 stupňová mikroarchitektura [DMIPS/MHz]**
Xilinx [15]	1,07	1,34
Codasip model	0,82	0,97

2.4.3 Funkční verifikace

Nejlepším ověřením správnosti návrhu je funkční verifikace, při které se ověřuje přímo generovaný HDL popis pomocí simulátorů HDL jazyků (Modelsim, Riviera aj.). Automaticky je generováno verifikační prostředí podle metodologie UVM (Universal Verification Methodology). Při funkční verifikaci je kontrolován HDL popis proti IA modelu procesoru.

U funkční verifikace se nekontroluje návratový kód programu, ale shoda registrů a pamětí po běhu programu. Kromě kontroly shody registrů a pamětí umožňuje funkční verifikace i počítání pokrytí HDL kódu procesoru. Při zjišťování pokrytí kódu se zaznamenává, které příkazy (řádky kódu) byly při testování vykonány, přechody mezi stavy stavových automatů, pokrytí všech kombinací logických výrazů atd.

Pro funkční verifikaci byla použita stejná testovací sada jako pro základní testování. Výsledky verifikace jsou shodné s výsledky simulace CA modelů. Výsledky pokrytí HDL kódu jsou uvedené v tabulce 2.7. Pokrytí kódu by mohlo být zvýšeno použitím více instrukcí (C překladač např. nevyužívá instrukce pro přístup ke speciálním registrům) nebo zahrnutím testování přerušení do funkční verifikace.

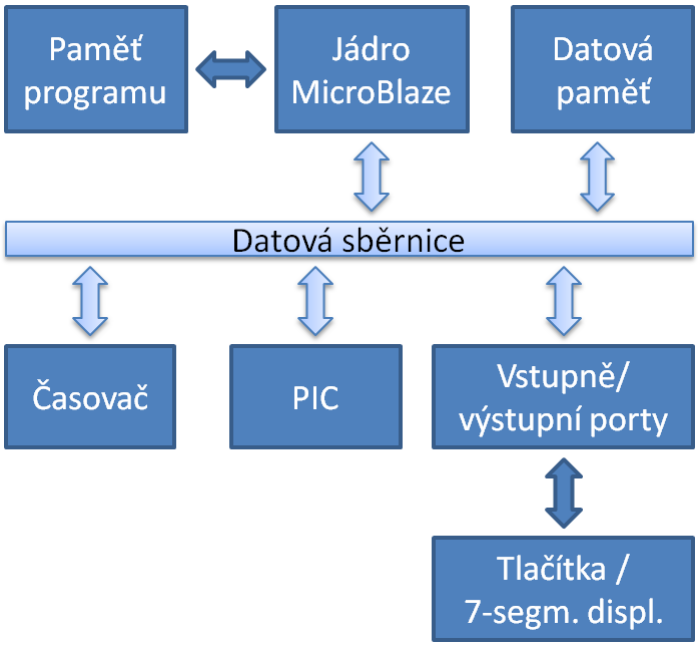
Tab. 2.7: Výsledky funkční verifikace - pokrytí kódu

Model	**pokrytí kódu**
3 stupňová mikroarchitektura	89,86 %
5 stupňová mikroarchitektura	90,05 %

Speciálně pro funkční verifikaci je k dispozici i nástroj, který generuje náhodné aplikace („randomgen“). Tento nástroj generuje náhodné programy v jazyce assembler, které mohou být efektivně využity pro testování a zvyšování pokrytí kódu. Z důvodu omezení tohoto nástroje však nelze pro MicroBlaze vygenerovat náhodné aplikace.

2.4.4 Funkční testování v FPGA

Po ověření funkčnosti navržených modelů pomocí funkční verifikace je možné procesor testovat přímo v obvodech FPGA. Jako praktická ukázka byla vytvořena aplikace digitálních hodin. Praktická ukázka využívá zapojení platformy dle obrázku 2.5.



Obr. 2.5: Zapojení procesoru MicroBlaze na platformě pro testování v obvodu FPGA

K procesoru MicroBlaze je přes datovou sběrnici připojena datová paměť, jednoduchý časovač, programovatelný řadič přerušení PIC (Programmable Interrupt Controller) a komponenta vstupně/výstupních portů. Časovač generuje přerušení, které je přes PIC předáno procesoru MicroBlaze. Na vstupní porty je připojeno ovládání pomocí tlačítek a přepínače testovací desky, na registrované výstupní porty je připojen řadič 7-segmentových displejů určených pro zobrazování času. Pro správnou funkci programu je kromě zapojení platformy nutné také upravit „startup“ kód tak, aby se správně vykonávalo volání a návrat z obsluhy přerušení.

Procesor přistupuje ke všem komponentám připojeným na datovou sběrnicí pomocí instrukcí pro načítání a ukládání do paměti. Po resetu je inicializován PIC a časovač tak, aby se generovalo přerušení s periodou 50 ms. V obsluze přerušení je při každém přerušení kontrolován stav přepínače a tlačítek. Při přepnutí přepínače program přejde do módu, ve kterém lze nastavit libovolný čas pomocí tlačítek. V běžném módu je stisk tlačítek ignorován a každé 20. přerušení je inkrementován čas o 1 s.

Konfigurační soubory byly vytvořeny pro testovací desku NEXYS 4 s obvodem FPGA Artix7 XC7A100T-1 v pouzdře CSG324. Tato deska byla zvolena vzhledem k dobré dostupnosti a vhodnému počtu 7-segmentových displejů pro aplikaci digitálních hodin. V případě portování na jiný obvod FPGA je nutné změnit konfiguraci pinů obvodu FPGA.

3 ZÁVĚR

V rámci diplomové práce byla popsána problematika návrhu procesorů s řetězenými linkami a popisu procesorů v jazyce CodAL, který je vyvíjen firmou Codasip. Práce dále seznamuje se základy architektury procesorového jádra MicroBlaze od firmy Xilinx, které je navrženo pro implementaci do obvodů FPGA.

V rámci praktické části byl v jazyce CodAL vytvořen model procesoru MicroBlaze na instrukční úrovni. Model byl doplněn o informace nutné pro generování překladače jazyka C. Implementovaná instrukční sada je s výjimkou instrukcí pro odčítání kompatibilní s instrukční sadou originálního jádra. Instrukce pro odčítání se z důvodu omezení překladače liší v přístupu k příznaku carry. K instrukční sadě procesoru bylo navíc přidáno několik instrukcí, které umožňují lepší ladění programů v simulátoru, zastavování běhu simulátoru a vyvolání přerušení.

K modelu na instrukční úrovni byly vytvořeny dva modely na RTL úrovni implementující třístupňovou a pětistupňovou linku procesoru včetně přerušení. Oba modely vycházejí z existujících informací o mikroarchitektuře procesoru MicroBlaze a to včetně časování. Třístupňová mikroarchitektura byla optimalizována na plochu, pětistupňová na frekvenci. Při porovnání výsledků syntézy bylo zjištěno, že vlastní implementace dosahuje nižší pracovní frekvence a většího počtu použitých zdrojů než vysoce optimalizovaná jádra od výrobce. Výhodou tohoto přístupu však může být libovolné rozšíření či úprava v mikroarchitektuře bez závislosti na výrobci nebo možnost implementace do obvodů ASIC.

Správná funkce modelů byla otestována na sadě více než 700 programů a sadě základních testů výkonnosti. Na základě testu Dhryston bylo zjištěno, že generovaný překladač nedosahuje výkonu jako překladač poskytovaný výrobcem. Generovaný překladač má několik známých omezení a další optimalizací nástrojů by pravděpodobně bylo dosaženo lepších výsledků. Po základním testování funkce procesoru byla provedena funkční verifikace včetně měření pokrytí HDL kódu. Během testování a verifikace nebyly odhaleny žádné chyby v návrhu.

Po ověření funkce verifikací byly oba modely otestovány i funkčně v obvodu FPGA. Za účelem testování byl vytvořen demonstrační příklad digitálních hodin s využitím přerušení procesoru. Správná funkce byla ověřena na vývojové desce NEXYS 4.

LITERATURA

[1] Procesor [online]. [cit. 2015-11-13]. Dostupné z URL: <http://www.outech-havirov.cz/skola/files/knihovna_eltech/epo/cpu.pdf>.

[2] OLIVKA, Petr. Procesory CISC a RISC: Studijní materiál pro předmět Architektury počítačů [online]. Ostrava, 2010 [cit. 2015-11-07]. Dostupné z URL: <http://poli.cs.vsb.cz/edu/arp/down/procrisc.pdf>.

[3] Mikroarchitektura. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2015-10-27]. Dostupné z URL: <https://cs.wikipedia.org/wiki/Mikroarchitektura>.

[4] Charakteristika dalších verzí procesorů v PC [online]. In: . [cit. 2015-11-26]. Dostupné z URL: <http://www.fit.vutbr.cz/study/courses/ITP/public/itp07/pentium01_09.pdf>.

[5] JOHN L. HENNESSY, John L.David A a WITH CONTRIBUTIONS BY ANDREA C. ARPACI-DUSSEAU .. [ET AL.]. Computer architecture a quantitative approach. 4th ed. Amsterdam: Elsevier/Morgan Kaufmann Publishers, 2007. ISBN 978-008-0475-028.

[6] Architektura procesorů (2): Pipelining a hazardy. In: POPULAR [online]. [cit. 2015-12-01]. Dostupné z URL: <http://popular.fbmi.cvut.cz/it/Stranky/Architektura-procesoru-2-Pipelining-a-hazardy.aspx>.

[7] BEČVÁŘ, Miloš. Proudové zpracování instrukcí II.; Hazardy v proudovém zpracování; Proudové zpracování FP instrukcí: Architektura počítačů [online]. [cit. 2015-12-03]. Dostupné z URL: <http://fel.jahho.cz/5.semestr/aps/prednasky/aps_pipeline_cpu2_1s.pdf>.

[8] STEIL, Michael. Having Fun with Branch Delay Slots. In: Pagetable.com: Some Assembly Required [online]. 2009 [cit. 2015-11-19]. Dostupné z URL: <http://www.pagetable.com/?p=313>.

[9] SHEN, John Paul a Mikko H LIPASTI. Modern processor design: fundamentals of superscalar processors. 1st ed. Boston: McGraw-Hill Higher Education, 2005, xiv, 642 p. ISBN 00-705-7064-7.

[10] MASAŘÍK, Karel. Jazyky pro popis architektury [online]. 2006 [cit. 2015-11-15]. Dostupné z URL: <http://www.fit.vutbr.cz/research/pubs/TR/2006/sem_uifs/s060306slidy1.pdf>.

[11] CODASIP. Codasip Studio User Guide. Verze 1.0.3. Brno, 2016.

[12] CODASIP. Codasip Studio Technical Reference Manual. Verze 1.0.4. Brno, 2016.

[13] CODASIP. CodAL Language Reference Manual. Verze 1.0.4. Brno, 2016.

[14] XILINX. MicroBlaze Processor Reference Guide: Embedded Development Kit EDK 14.7 [online]. Brno, 2013 [cit. 2016-02-15]. Dostupné z URL: <http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/mb_ref_guide.pdf>.

[15] MicroBlaze Soft Processor Core [online]. 2015 [cit. 2016-05-06]. Dostupné z URL: <http://www.xilinx.com/products/design-tools/microblaze.html>.

[16] Wikipedie: Otevřená encyklopedie: Carry flag. [online]. c2015 [cit. 2016-03-27]. Dostupné z URL: <https://en.wikipedia.org/wiki/Carry_flag>.

[17] LLVM Overview. The LLVM Compiler Infrastructure [online]. [cit. 2016-04-27]. Dostupné z URL: <www.llvm.org>.

[18] LEVINE, John R. Linkers and loaders. San Francisco: Morgan Kaufmann, c2000. ISBN 1558604960.

[19] WEICKER, Reinhold P. Dhrystone: a synthetic systems programming benchmark. Communications of the ACM. 27(10), 1013-1030. DOI: 10.1145/358274.358283. ISSN 00010782. Dostupné také z URL: <http://portal.acm.org/citation.cfm?doid=358274.358283>.

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

ABI	Application Binary Interface, nízkoúrovňové rozhraní aplikací
ADL	Architecture Description Language, jazyk pro popis architektur
ALU	Arithmetic-Logic Unit, aritmeticko-logická jednotka
ASIC	Application-Specific Integrated Circuit, zákaznický integrovaný obvod
ASIP	Application Specific Instruction-set Processor, procesor s aplikačně specifickou instrukční sadou
AXI	Advanced eXtensible Interface, pokročilé rozšiřitelné rozhraní
BIP	Break In Progress, příznak provádění obsluhy instrukce break
BRAM	Block Random-Access Memory, bloková paměť s přímým přístupem
BS	Barrel Shifter, jednotka pro vícebitový posun dat
BTC	Branch Target Cache, dočasná paměť pro adresy skoků
BTR	Branch Target Register, registr s adresou cíle skoku
CA	Cycle Accurate, (model) na úrovni RTL
CC	Carry Copy, kopie příznaku přetečení (carry)
CISC	Complete Instruction Set Computing, procesor s kompletní instrukční sadou
CLB	Codasip Local Bus, název sběrnice
CodAL	Codasip Architecture Language, jazyk pro popis architektury
CPU	Central Processing Unit, centrální procesorová jednotka
DIV	Division, instrukce dělení, příznak v PVR registru
DS	Delay Slot, instrukce za skokem, která se vykonává vždy
DSP	Digital Signal Processing, jednotka pro digitální zpracování signálů
EAR	Exception Address Register, registr s adresou výjimky
EC	Exit Code, návratový kód
EE	Exception Enable, příznak povolení výjimek

EIP	Exception In Progress, příznak probíhající obsluhy výjimky
ESR	Exception Status Register, stavový registr pro výjimky
ESS	Exception Specific Statu, specifický kód výjimky
EX	Execute, stupeň linky - vykonávání instrukce
FE	Fetch, stupeň linky - načítání instrukce
FPGA	Field Programmable Gate Array, programovatelné hradlové pole
FPU	Floating Point Unit, jednotka pro zpracování výpočtů v plovoucí desetinné čárce
HDL	Hardware Description Language, jazyk pro popis hardware
IA	Instruction Accurate, (model) na instrukční úrovni
ID	Instruction Decode, stupeň linky - dekódování instrukce
IE	Interrupt Enable, příznak povolení přerušení
IF	Instruction Fetch, stupeň linky - načítání instrukce
IP	Intelectual Property, duševní vlastnictví
ISA	Instruction Set Architecture, instrukční sada
LAU	Least Addressable Unit, nejmenší adresovatelná jednotka
LMB	Local Memory Bus, název sběrnice
LUT	Look-Up Table, náhledová tabulka
MBV	MicroBlaze (release) Version, konstanta určující verzi procesoru MicroBlaze
MEM	Memory, stupeň linky - přístup do paměti
MFS	Move From Special register, instrukce pro čtení ze speciálních registrů
MIPS	Million Instructions Per Second, jednotka rychlost - milion zpracovaných instrukcí za sekundu
MMU	Memory Management Unit, jednotka správy paměti
MSR	Machine Status Register, stavový registr

MTS	Move To Special register, instrukce pro zápis do speciálních registrů
MUL	Multiplication, instrukce násobení, příznak v PVR registru
OOO	Out Of Order, zpracování mimo pořadí
PC	Program Counter, programový čítač
PIC	Programmable Interrupt Controller, programovatelný řadič přerušení
PLB	Processor Local Bus, název sběrnice
PVR	Processor Version Register, registr s označením verze procesoru
RAW	Read After Write, hazard „čtení po zápise“
RISC	Reduced Instruction Set Computing, procesor s redukovanou instrukční sadou
RTL	Register Transfer Level, popis na úrovni přenosu signálu s využitím registrů
RW	Read/Write, čtení/zápis
SIMD	Single Instruction Multiple Data, vektorový procesor
SoC	System on Chip, systém na čipu
SW	Software, programové vybavení
TO	Time-Out, časový limit
UVM	Universal Verification Methodology, univerzalní verifikační metodologie
VHDL	VHSIC Hardware Description Language, jazyk pro popis velmi rychlých integrovaných obvodů
VHSIC	Very High Speed Integrated Circuits, velmi rychlé integrované obvody
VLIW	Very Long Instruction Word, procesory s dlouhým instrukčním slovem
WAR	Write After Read, hazard „zápis po čtení“
WAW	Write After Write, hazard „zápis po zápise“
WB	Writeback, stupeň linky - zápis do registrů

SEZNAM PŘÍLOH

A SEZNAM IMPLEMENTOVANÝCH INSTRUKCÍ

Tab. A.1: Seznam implementovaných instrukcí [14]

Syntax	Sémantika
ADD Rd,Ra,Rb	Rd = Rb + Ra MSR.C = carry(Rb, Ra)
ADDC Rd,Ra,Rb	Rd = Rb + Ra + MSR.C MSR.C = carry(Rb, Ra, MSR.C)
ADDI Rd,Ra,imm	Rd = imm + Ra MSR.C = carry(imm, Ra)
ADDIC Rd,Ra,imm	Rd = imm + Ra + MSR.C MSR.C = carry(imm, Ra, MSR.C)
ADDIK Rd,Ra,imm	Rd = imm + Ra
ADDIKC Rd,Ra,imm	Rd = imm + Ra + MSR.C
ADDK Rd,Ra,Rb	Rd = Rb + Ra
ADDKC Rd,Ra,Rb	Rd = Rb + Ra + MSR.C
AND Rd,Ra,Rb	Rd = Ra & Rb
ANDI Rd,Ra,imm	Rd = Ra & imm
ANDN Rd,Ra,Rb	Rd = Ra & $\overline{Rb}$
ANDNI Rd,Ra,imm	Rd = Ra & $\overline{imm}$
BEQ Ra,Rb	if (Ra == 0) PC = PC + Rb
BEQD Ra,Rb	if (Ra == 0) PC = PC + Rb (delay slot)
BEQI Ra,imm	if (Ra == 0) PC = PC + imm
BEQID Ra,imm	if (Ra == 0) PC = PC + imm (delay slot)
BGE Ra,Rb	if (Ra >= 0) PC = PC + Rb
BGED Ra,Rb	if (Ra >= 0) PC = PC + Rb (delay slot)
BGEI Ra,imm	if (Ra >= 0) PC = PC + imm
BGEID Ra,imm	if (Ra >= 0)PC = PC + imm (delay slot)
BGT Ra,Rb	if (Ra > 0) PC = PC + Rb
BGTD Ra,Rb	if (Ra > 0) PC = PC + Rb (delay slot)
BGTI Ra,imm	if (Ra > 0) PC = PC + imm
BGTID Ra,imm	if (Ra > 0) PC = PC + imm (delay slot)
BLE Ra,Rb	if (Ra <= 0) PC = PC + Rb
BLED Ra,Rb	if (Ra <= 0) PC = PC + Rb (delay slot)
BLEI Ra,imm	if (Ra <= 0) PC = PC + imm
BLEID Ra,imm	if (Ra <= 0) PC = PC + imm (delay slot)

Syntax	Sémantika
BLT Ra,Rb	if (Ra < 0) PC = PC + Rb
BLTD Ra,Rb	if (Ra < 0) PC = PC + Rb (delay slot)
BLTI Ra,imm	if (Ra < 0) PC = PC + imm
BLTID Ra,imm	if (Ra < 0) PC = PC + imm (delay slot)
BNE Ra,Rb	if (Ra != 0) PC = PC + Rb
BNED Ra,Rb	if (Ra != 0) PC = PC + Rb (delay slot)
BNEI Ra,imm	if (Ra != 0) PC = PC + imm
BNEID Ra,imm	if (Ra != 0) PC = PC + imm (delay slot)
BR Rb	PC = PC + Rb
BRA Rb	PC = Rb
BRAD Rb	PC = Rb (delay slot)
BRAI imm	PC = imm
BRAID imm	PC = imm (delay slot)
BRALD Rd,Rb	Rd = PC PC = Rb (delay slot)
BRALID Rd,imm	Rd = PC PC = imm (delay slot)
BRD Rb	PC = PC + Rb (delay slot)
BRI imm	PC = PC + imm
BRID imm	PC = PC + imm (delay slot)
BRLD Rd,Rb	Rd = PC PC = PC + Rb (delay slot)
BRLID Rd,imm	Rd = PC PC = PC + imm (delay slot)
BSLL Rd,Ra,Rb	Rd = Ra « (Rb & 0x1F)
BSLLI Rd,Ra,imm	Rd = Ra « (imm & 0x1F)
BSRA Rd,Ra,Rb	Rd = s(Ra » (Rb & 0x1F))
BSRAI Rd,Ra,imm	Rd = s(Ra » (imm & 0x1F))
BSRL Rd, Ra,Rb	Rd = u(Ra » (Rb & 0x1F))
BSRLI Rd,Ra,imm	Rd = u(Ra » (imm & 0x1F))
CLZ Rd, Ra	Rd = count_leading_zeroes(Ra)
CMP Rd,Ra,Rb	Rd = Rb - Ra if (Rb (s)>= Ra) Rd[31] = 0 else Rd[31] = 1
CMPU Rd,Ra,Rb	Rd = Rb - Ra if (Rb (u)>= Ra) Rd[31] = 0 else Rd[31] = 1

Syntax	Sémantika
IMM imm_hi [1]	imm[31..16] = imm_hi
LBU Rd,Ra,Rb	Rd = (uint8)dmem[Ra + Rb]
LBUI Rd,Ra,imm	Rd = (uint8)dmem[Ra + imm]
LHU Rd,Ra,Rb	Rd = (uint16)dmem[Ra + Rb]
LHUI Rd,Ra,imm	Rd = (uint16)dmem[Ra + imm]
LW Rd,Ra,Rb	Rd = (uint32)dmem[Ra + Rb]
LWI Rd,Ra,imm	Rd = (uint32)dmem[Ra + imm]
MFS Rd,Sa	Rd = SPR[Sa]
MSRCLR Rd,imm	Rd = MSR MSR = MSR & $\overline{imm14}$
MSRSET Rd,imm	Rd = MSR MSR = MSR | imm14
MTS Sd,Ra	SPR[Sd] = Ra
MUL Rd,Ra,Rb	Rd = (int)Ra * (int)Rb
MULH Rd,Ra,Rb	Rd = ((int)Ra * (int)Rb) » 32
MULHSU Rd,Ra,Rb	Rd = ((int)Ra * (uint)Rb) » 32 (signed)
MULHU Rd,Ra,Rb	Rd = ((uint)Ra * (uint)Rb) » 32
MULI Rd,Ra,imm	Rd = (int)Ra * (int)imm
OR Rd,Ra,Rb	Rd = Ra | Rb
ORI Rd,Ra,imm	Rd = Ra | imm

PCMPBF Rd,Ra,Rb	if (Rb[31..24] = Ra[31..24]) Rd = 1 else
	if (Rb[23..16] = Ra[23..16]) Rd = 2 else
	if (Rb[15..8] = Ra[15..8]) Rd = 3 else
	if (Rb[7..0] = Ra[7..0]) Rd = 4 else
	Rd = 0

PCMPEQ Rd,Ra,Rb	if (Rd == Ra) Rd = 1 else Rd = 0
PCMPNE Rd,Ra,Rb	if (Rd != Ra) Rd = 1 else Rd = 0
RSUB Rd,Ra,Rb [2]	Rd = Rb - Ra MSR.C = borrow(Rb, Ra)
RSUBC Rd,Ra,Rb [2]	Rd = Rb - Ra - MSR.C MSR.C = borrow(Rb, Ra, MSR.C)
RSUBI Rd,Ra,imm [2]	Rd = imm - Ra MSR.C = borrow(imm, Ra)
RSUBIC Rd,Ra,imm [2]	Rd = imm - Ra - MSR.C MSR.C = borrow(imm, Ra, MSR.C)

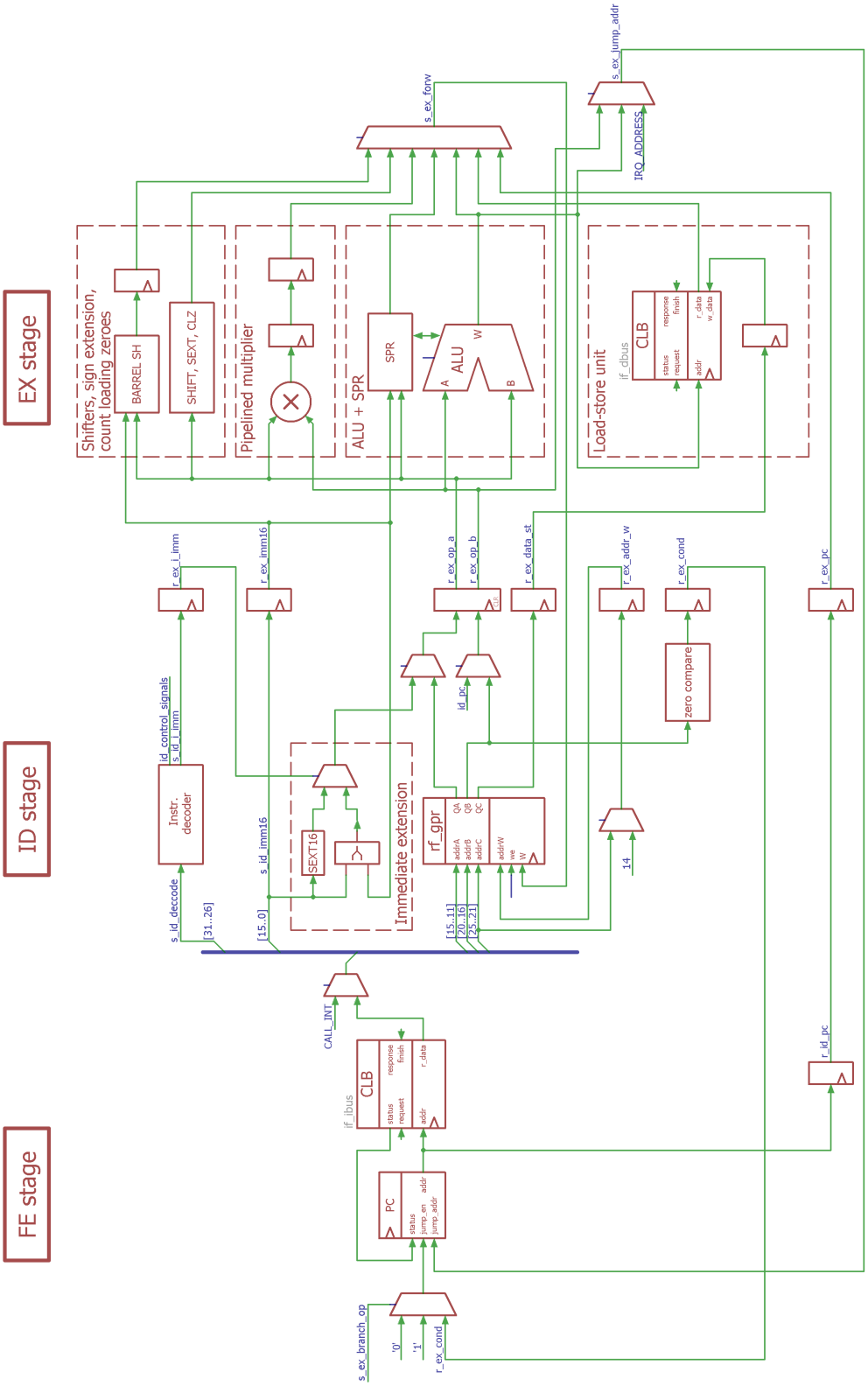
[1] Konstanta následující instrukce je rozšířená konstantou imm_hi jako {imm_hi, imm}

[2] Implementováno jako odčítání s borrow, originální jádro používá odčítání s carry

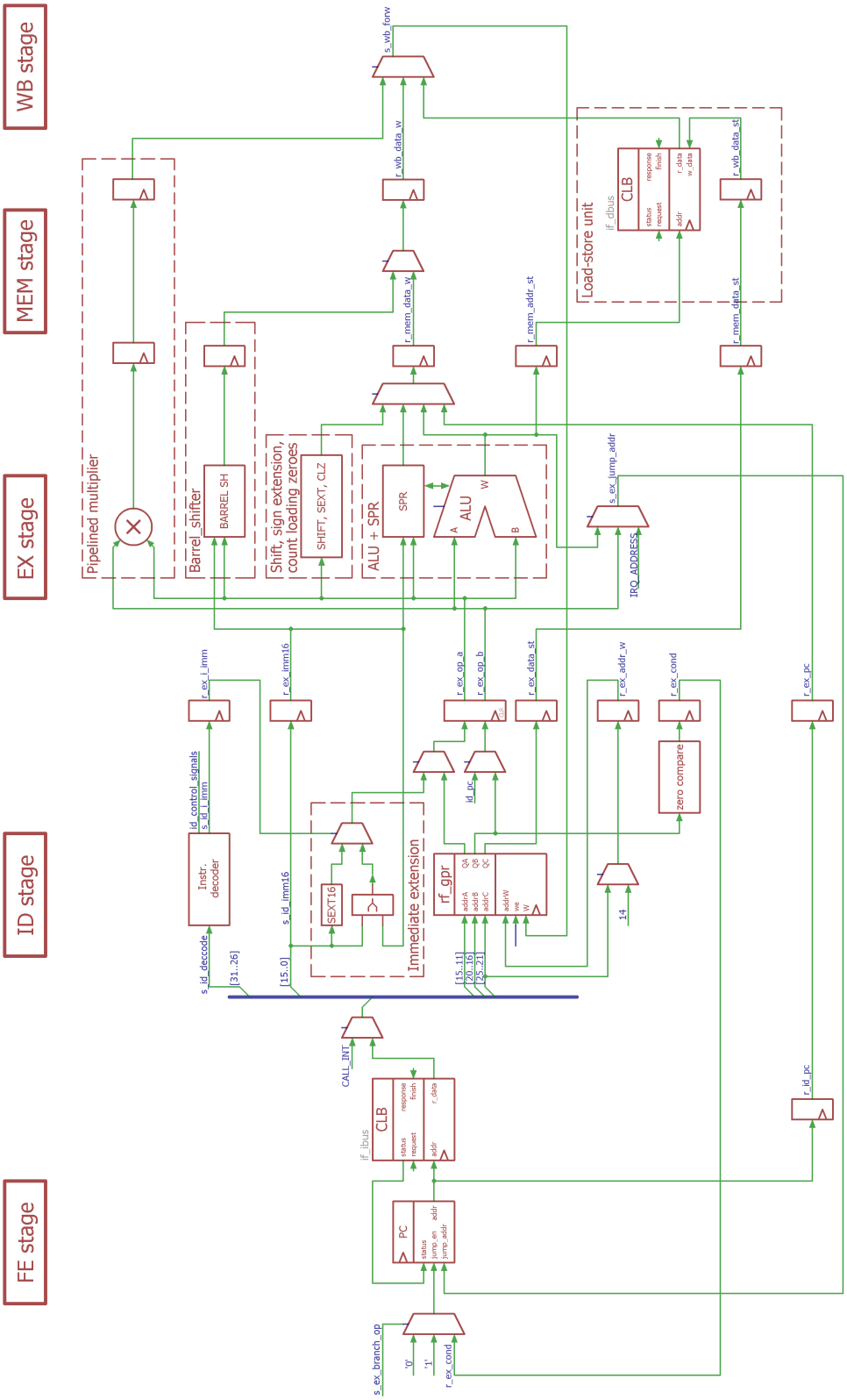
Syntax	Sémantika
RSUBIK Rd,Ra,imm	Rd = imm - Ra
RSUBIKC Rd,Ra,imm [2]	Rd = imm - Ra - MSR.C
RSUBK Rd,Ra,Rb	Rd = Rb - Ra
RSUBKC Rd,Ra,Rb [2]	Rd = Rb - Ra - MSR.C
RTID Ra,imm	PC = Ra + imm (delay slot) MSR.IE = 1
RTSD Ra,imm	PC = Ra + imm (delay slot)
SB Rd,Ra,Rb	dmem[Ra + Rb] = (uint8)Rd
SBI Rd,Ra,imm	dmem[Ra + imm] = (uint8)Rd
SEXT16 Rd,Ra	Rd = s(Ra[15..0])
SEXT8 Rd,Ra	Rd = s(Ra[23..0])
SH Rd,Ra,Rb	dmem[Ra + Rb] = (uint16)Rd
SHI Rd,Ra,imm	dmem[Ra + imm] = (uint16)Rd
SRA Rd,Ra	Rd = s(Ra » 1) MSR.C = Ra[0]
SRC Rd,Ra	Rd = {C, (Ra » 1)} MSR.C = Ra[0]
SRL Rd,Ra	Rd = u(Ra » 1) MSR.C = Ra[0]
SW Rd,Ra,Rb	dmem[Ra + Rb] = Rd
SWI Rd,Ra,imm	dmem[Ra + imm] = Rd
XOR Rd,Ra,Rb	Rd = Ra ⌢ Rb
XORI Rd,Ra,imm	Rd = Ra ⌢ imm
HALT	Zastaví simulaci
SYSCALL	Volání systémových funkcí
CALL_INT	R14 = PC PC = 0x10 MSR.IE = 0

[2]Implementováno jako odčítání s borrow, originální jádro používá odčítání s carry

B SCHÉMATA MIKROARCHITEKTURY



Obr. B.1: Schéma třístupňové linky



Obr. B.2: Schéma pětistupňové linky